Java
desde cero

Programación orientada a objetos + Concepto de clases con Java + Interpretación de código + Excepciones y genéricos + Manejo de errores + Librerías base

JAVA DESDE CERO

TÍTULO: Java desde cero

AUTOR: Ignacio Vivona

COLECCIÓN: Desde Cero

FORMATO: 19 x 15 cm

PÁGINAS: 192

ISBN 978-987-1949-68-7

Vivona, Ignacio

Java desde cero / Ignacio Vivona ; coordinado por Gustavo Carballeiro.

1a ed. - Ciudad Autónoma de Buenos Aires : Fox Andina; Buenos Aires: Dalaga, 2014.

192 p. ; 19x15 cm. - (Desde cero; 38)

ISBN 978-987-1949-68-7

1. Informática. I. Carballeiro, Gustavo, coord. II. Título

CDD 005.3

Prólogo al contenido

El lenguaje de programación Java está posicionado en el mercado desde hace tiempo, y se consagra como uno de los más importantes y poderosos. La ventaja de este lenguaje, por un lado, se encuentra en que está orientado a objetos; además, es simple de utilizar y multiplataforma.

Normalmente, un programa viene definido por un código fuente. Dicho código, al ser compilado, se transforma en una serie de sentencias escritas en lenguaje máquina que da lugar a interpretar alguna ejecución indicada por lo desarrollado.

Otra característica que tienen los lenguajes de programación es el manejo de la memoria. Existe un proceso independiente denominado Garbage Collector, que se encarga de liberar automáticamente toda la memoria que ya no se utiliza, de manera que la liberación de memoria se hace trasparente al programador.

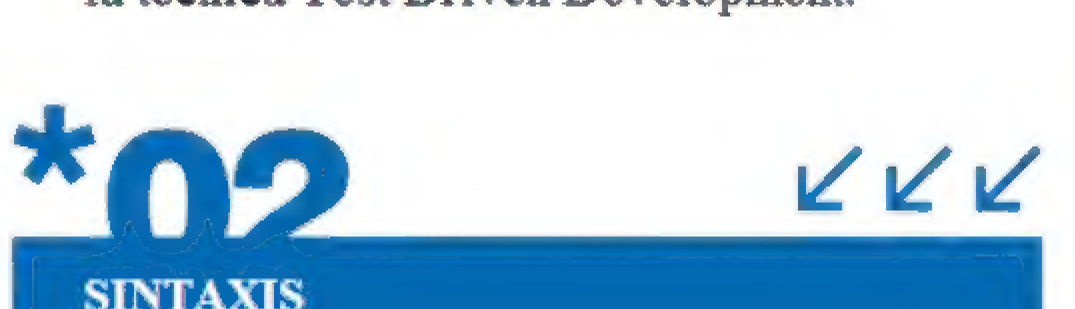
Una de las características que más potencia aporta a Java es que viene acompañado de una serie de librerías estándar para realizar una multitud de operaciones comunes a la hora de programar. Es el llamado Java API, que incluye tres bloques básicos: Java Standard Edition (JSE), Java Enterprise Edition (JEE) y Java Micro Edition (JME). La versión Standard cubre el desarrollo de aplicaciones de propósito general, y es la base sobre la que se apoyan las otras dos. La versión Enterprise proporciona librerías para el desarrollo de aplicaciones empresariales multicapa (ofrece los estándares para el desarrollo de aplicaciones en servidor), y la versión Micro está especialmente orientada a dispositivos embebidos, como teléfonos móviles, PDAs, etcétera.

Así, el mundo Java se abre ante nosotros, de quienes depende tomar el conocimiento que nos brinda este libro para volvernos desarrolladores de esta potente herramienta.

El libro de un vistazo

A lo largo de esta obra conoceremos las técnicas y herramientas indispensables para comenzar a programar en Java. Comenzaremos con los conceptos fundamentales de la programación orientada a objetos, el diseño y el desarrollo de software. Luego, iremos exponiendo los procedimientos con código fuente de ejemplo y diagramas conceptuales.

Antes de introducirnos por completo en la programación Java, necesitamos conocer su historia y las fuerzas que guiaron su creación. En este primer capítulo veremos también una introducción a la forma de trabajo que adoptaremos a lo largo del libro y conoceremos la técnica Test Driven Development.



La sintaxis es el conjunto de reglas que dan como resultado un programa adecuadamente desarrollado. Describen qué elementos están permitidos en determinados contextos y cómo se relacionan. En este capítulo, conoceremos la sintaxis básica necesaria para dominar la programación en Java.

Las clases son los moldes para la creación de objetos, ya que definen la forma y el comportamiento de las instancias creadas. A lo largo de este capítulo, veremos cómo se crean y cómo se definen sus atributos y métodos.

Existen mecanismos para reutilizar código sin caer en las trampas de la herencia múltiple. Un ejemplo son las interfaces. Al terminar este tema, conoceremos estructuras de código que nos serán útiles para realizar el desarrollo de nuestras aplicaciones: las enumeraciones.

*05

EXCEPCIONES Y GENÉRICOS

Cuando nuestro programa está siendo ejecutado y ocurre alguna situación inesperada, el sistema lo notifica mediante eventos denominados excepciones. Las excepciones son la forma de avisar y detectar errores en la ejecución o acontecimientos extraños.

*06

LIBRERÍA BASE

Quien desea aprender Java debe conocer las clases e interfaces que se proveen desde la instalación. No solo se ahorra trabajo al utilizar algo ya existente, sino que el código puede comunicar mejor su propósito a otros programadores, ya que usa elementos estándares.

*07

ANOTACIONES

Muchos sistemas y librerías requieren de una gran cantidad de información suplementaria al código para poder funcionar. Las anotaciones son un mecanismo para poder unificar toda esta información en una forma estandarizada.

*Ap ON WEB

TÉCNICAS Y DISEÑO

Para finalizar el aprendizaje, debemos saber de qué manera combinar los elementos del lenguaje para construir programas claros, extensibles y flexibles. En este apéndice, conoceremos ciertos conceptos de diseño que nos resultarán muy útiles.

INFORMACIÓN COMPLEMENTARIA

A lo largo de este manual, podrá encontrar una serie de recuadros que le brindarán información complementaria: curiosidades, trucos, ideas y consejos sobre los temas tratados. Para que pueda distinguirlos en forma más sencilla, cada recuadro está identificado con diferentes iconos:

CURIOSIDADES E IDEAS

ATENCIÓN

DATOS ÚTILES Y NOVEDADES

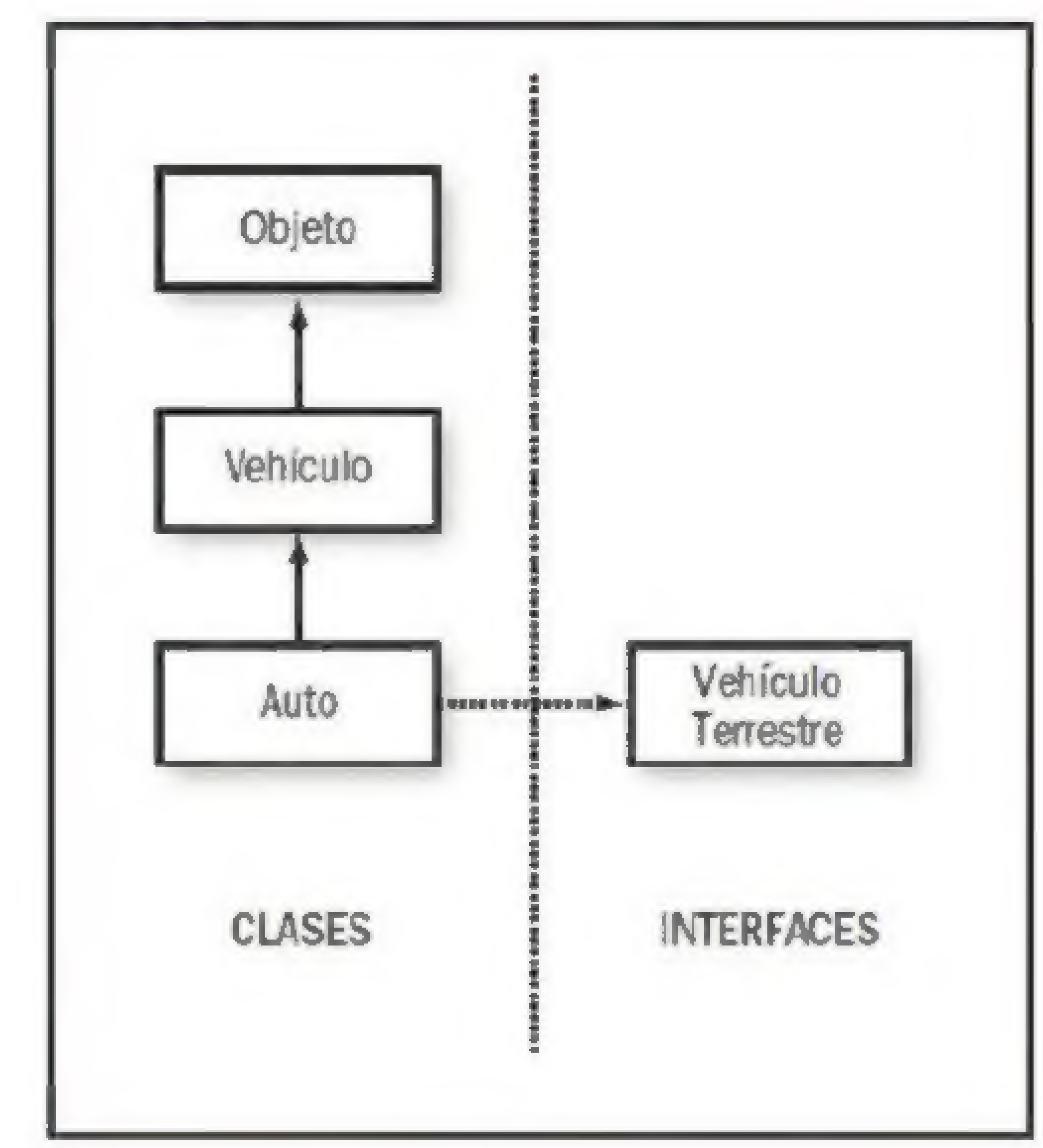
SITIOS WEB

Contenido del libro

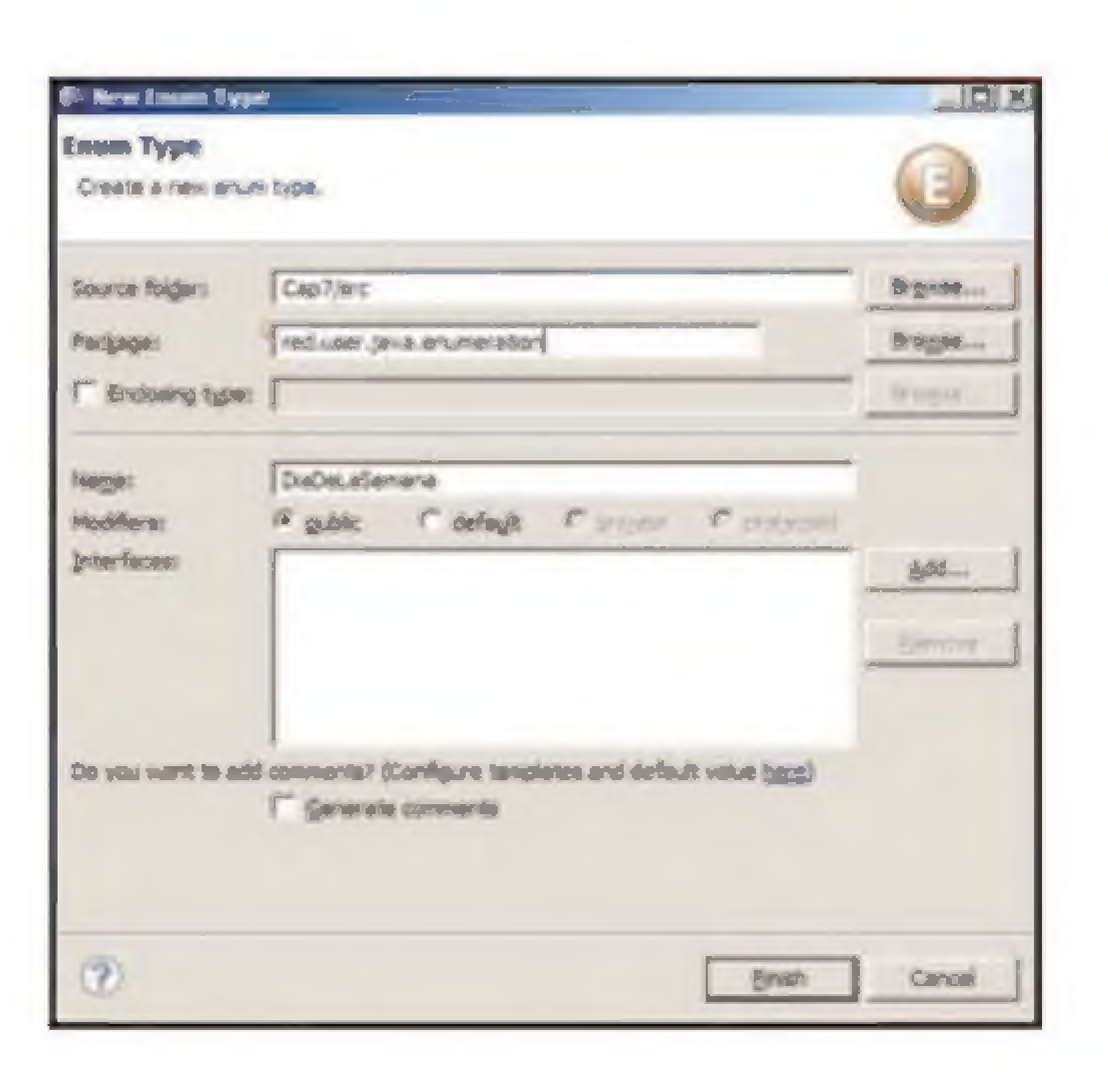
Descargas de Software

Desarrolladores:

Clientes:



Interfaces y enumeraciones

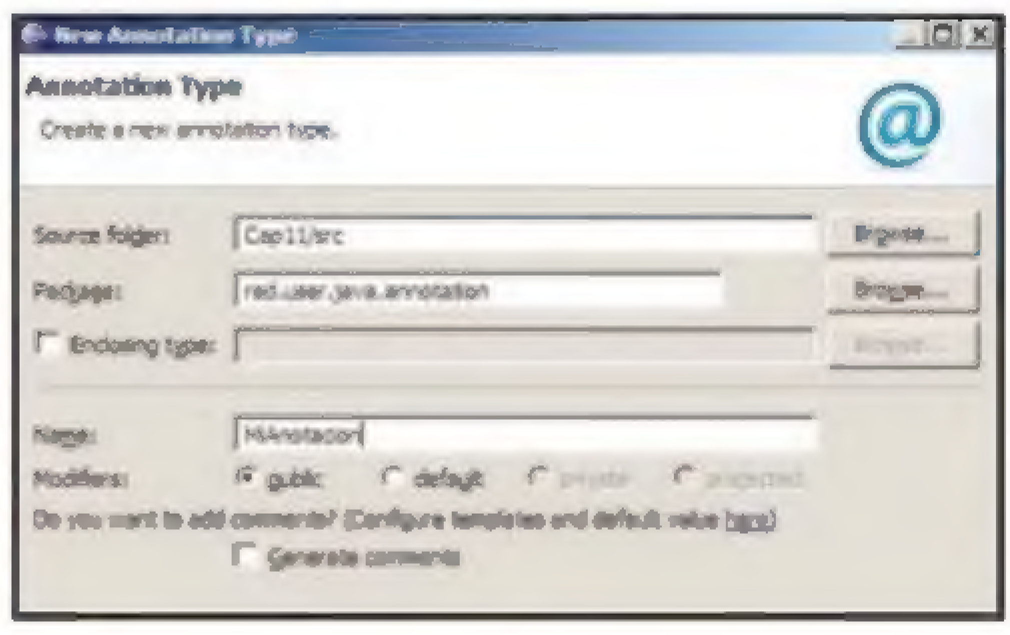


Excepciones y genéricos

Librería base

*07

Anotaciones



Técnicas y diseño

Introducción

¿Cómo funciona esto? ¿Es posible hacer algo así? Estas son preguntas que todo aquel que se interesa por la tecnología y la computación se hace. Y estas inquietudes son las que nos llevan a embarcarnos en el camino de la programación. Hoy en día, las computadoras están en todos lados y en todas partes.

Cualquiera tiene los medios y la posibilidad de responder las dudas anteriores y de crear programas y herramientas que permitan mejorar el mundo. Cualquiera puede ser el autor de la próxima gran idea millonaria.

La programación orientada a objetos ha dominado el mercado por sobre otros paradigmas durante los últimos años, y parece que va seguir dominándolo por mucho tiempo más. En particular, Java es el máximo referente de los programadores y existe una comunidad que constantemente genera productos, librerías y frameworks. Aun así, la demanda de nuevos programadores sigue creciendo. La orientación a objetos encabeza la difícil tarea de transmitir ideas y de plasmarlas en la PC, de tal forma que un sistema pueda hacer lo que queramos de una manera sencilla y tangible.

En este libro, el lector tendrá la oportunidad de sumergirse en el paradigma; tomar sus ideas, sus reglas y sus metodologías.

Luego, todos estos conocimientos se aplicarán a un lenguaje particular; en este caso, Java. Irá conociendo y aprendiendo el lenguaje paso a paso, desde su sintaxis y semántica, para escribir código correcto y entender cómo plasmar y modelar adecuadamente situaciones de su interés. Para esto, se usarán las herramientas provistas por el lenguaje, como las clases y la herencia, y también las interfaces y conceptos que provee Java. El lector está invitado a continuar, con la mente abierta a nuevas ideas, sin preconceptos ni prejuicios.

Iniciación a Java

Antes de ingresar por completo en la programación Java necesitamos conocer su historia y las fuerzas que guiaron su creación. Veremos también una introducción a la forma de trabajo que adoptaremos a lo largo del libro y haremos un breve vistazo a la técnica de Test Driven Development.

Historia

El lenguaje de programación **Java** tiene sus orígenes en el año 1991, cuando **Sun** empieza el proyecto **Green**. Este proyecto tenía como objetivo controlar dispositivos hogareños, para lo cual se creó un lenguaje llamado **Oak**, cuyo nombre fue modificado por el actual, Java, en 1995.

Inicialmente, Java se lanzó como un lenguaje cuyos programas podían ejecutarse en cualquier plataforma. El slogan de Sun era " **write once, run anywhere**" ("escribir una vez, correrlo en cualquier parte"). Para lograrlo, Java corre sobre una **máquina virtual** (**virtual machine**) o un programa que simula una máquina abstracta, la cual funciona aislando al programa que corre sobre ella de los distintos hardwares y sistemas operativos. De esta forma, para el programa, la máquina donde corre es siempre igual.

James Gosling, padre de Java, quiso hacerlo parecido a **C++** para que los programadores de este lenguaje se sintieran cómodos con Java y optaran por él. Java se presentaba como un lenguaje parecido a C++ pero simplificado y con un manejo automático de memoria (es decir, el programador no era responsable de liberar la memoria no utilizada). De esto se encargaría una función de la máquina virtual llamada **recolector de basura** (**garbage collector**), un proceso que se ejecuta paralelamente al de la aplicación y se encarga de liberar la memoria ocupada por los objetos que no son utilizados por el sistema.

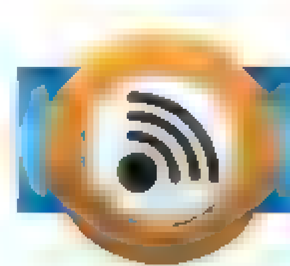
Para obtener material adicional gratuito, ingrese a la sección Publicaciones/Libros dentro de **http://premium.redusers.com**. Allí encontrará todos nuestros títulos y podrá acceder a contenido extra de cada uno, como sitios web relacionados, programas recomendados, ejemplos utilizados por el autor, apéndices y archivos editables o de código fuente. Todo esto ayudará a comprender mejor los conceptos desarrollados en la obra.

Java triunfó gracias a internet. En un comienzo lo hizo a través de las **applets**, pequeñas aplicaciones embebidas en las páginas web que se ejecutan en los navegadores, que no triunfaron dado que competían con la tecnología **Shockwave** (ahora conocida como **Flash**). A partir de 1997, Java empezó a migrar hacia los servidores. Se lanzó la primera versión de los **Servlets** y fue en esta área donde el lenguaje se impuso y brilló.

> JAVA COMENZÓ A IMPONERSE CON LA PRIMERA VERSIÓN DE LOS SERVLETS

Las tecnologías competidoras de ese entonces eran **CGI**, **ASP** y **PHP**. Java superó a la competencia y se posicionó como la tecnología por excelencia para las aplicaciones web empresariales. Desde entonces, es una de las tecnologías más importantes, disponible en casi todas las plataformas. Se encuentra en las desktops (aplicaciones de escritorio), en servidores (páginas web y aplicaciones web) y hasta en los dispositivos móviles (versión micro de Java).

Hoy, Java forma parte de uno de los sistemas operativos para celular más famosos y pujantes: **Android**. Este sistema fue creado por los desarrolladores de **Google** y está basado en Java, por lo cual, todas las aplicaciones para celulares (y también netbooks y tablets) con Android están hechas con este lenguaje. En el 2009, **Oracle**, gigante de las bases de datos, compró Sun y, por lo tanto, también asumió la responsabilidad sobre la evolución de Java.



GARBAGE COLLECTOR

Los lenguajes que no utilizan este mecanismo para administrar la memoria, como **C++**, requieren que el programador libere manualmente la memoria que no se va a utilizar más. Esta es la causa de numerosos errores en los programas y del consumo de mucha memoria. Aquella que no se libera, pero tampoco se usa, se conoce como **memory leak**.

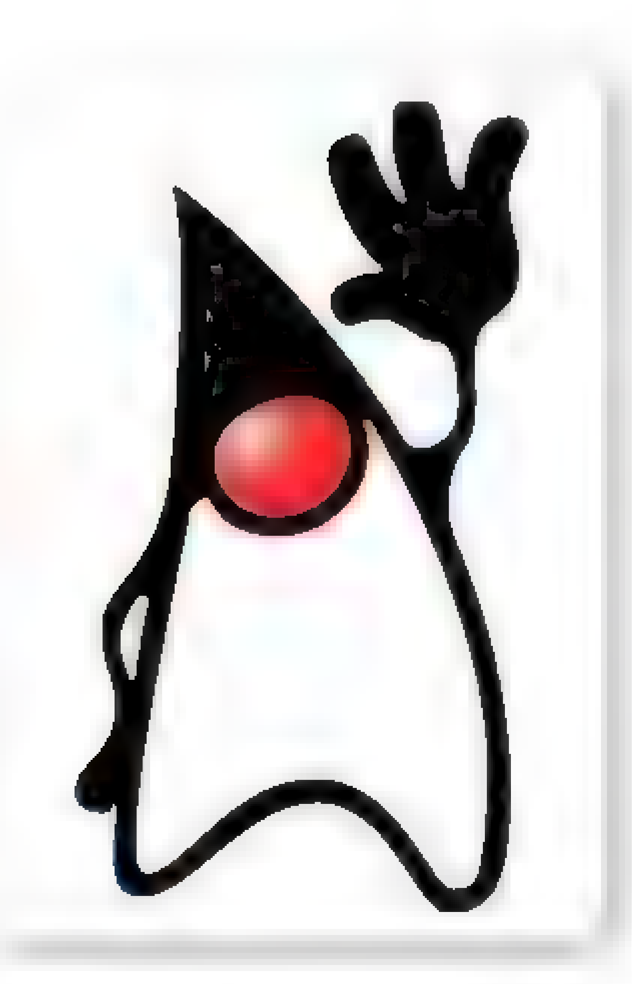


Figura 1. **Duke**, la mascota de Java en los comienzos del lenguaje, fue muy popular entre los programadores. Se la veía en cada artículo, libro o sitio relacionado con este lenguaje.

Requisitos para empezar a programar

Para poder programar en Java, compilar y correr los programas, necesitaremos instalar el **kit de desarrollo**, que podremos conseguir en la sección de descargas de Oracle.

Figura 2. Podemos encontrar amplia variedad de herramientas en la página de descargas de Oracle.

Debemos asegurarnos de descargar la última versión del **SDK**, *Software Development Kit* (Kit de Desarrollo de Software), y no el **JRE**, *Java Runtime Environment* (Entorno de Ejecución Java). El primero nos permitirá crear y ejecutar aplicaciones Java, mientras que el segundo solamente correrlas.

Finalizada la instalación, tendremos a nuestra disposición los distintos programas para producir y ejecutar aplicaciones Java, entre los cuales se encuentran el compilador javac.exe (que produce los binarios Java a partir del código fuente), el intérprete java.exe (que ejecuta un programa Java) y el documentador javadoc.exe (que genera la documentación de las clases). También encontraremos el *profiler* jvisualvm.exe, una herramienta que nos permitirá medir la performance (tiempo de ejecución y uso de memoria) de cualquier programa Java, de una forma sencilla y visual. También contaremos con las librerías base que trae Java por defecto y el código fuente de estas, por si queremos ver cómo funcionan internamente.

Eclipse IDE

Ya estamos en condiciones de empezar a programar nuestro primer código. Podríamos comenzar escribiendo con cualquier editor de texto y luego utilizar el **compilador** (programa que toma un archivo con código fuente, lo transforma y genera otro que entiende la máquina virtual), pero esta forma de trabajar solo sirve para algunas pruebas sencillas y no para desarrollos profesionales.

Tanto en la página donde descargamos el SDK como dentro de este, encontraremos ejemplos y tutoriales que nos ayudarán a profundizar los conocimientos que incorporaremos a lo largo del libro. Vale la pena mirar los ejemplos incluidos y probarlos, ya que utilizan muchas clases importantes de la librería base.

Por lo tanto, también necesitaremos descargar un entorno de trabajo: utilizaremos el **IDE** (*Integrated Development Environment* o entorno de desarrollo integrado) llamado **Eclipse**.

En el sitio **www.eclipse.org** encontraremos los enlaces para descargar el entorno de desarrollo más famoso. Allí encontraremos opciones para poder desarrollar en tecnologías diversas, desde Java, C++ y PHP, hasta JavaScript, y también numerosas herramientas para agregarle a nuestro IDE.

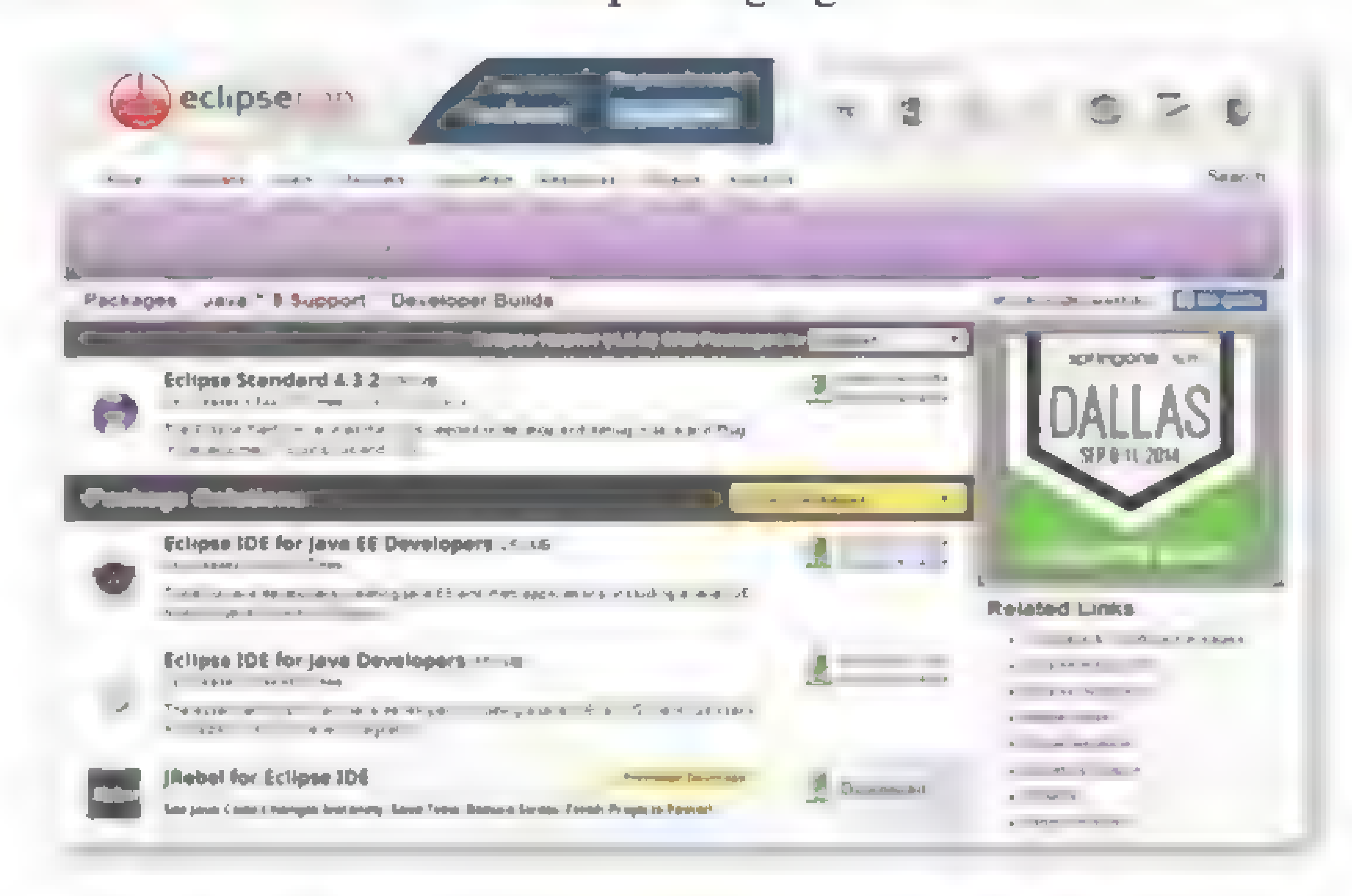


Figura 3. Página principal de Eclipse, donde podemos encontrar varias herramientas para desarrollar.

Eclipse es el entorno de desarrollo Java por excelencia. Fue concebido como una plataforma para la creación de IDEs, cuya expansión puede ser realizada mediante plugins. Inicialmente, los lenguajes soportados eran Java y luego C++; hoy existe una amplia variedad de plugins para casi todos los lenguajes, y los lenguajes nuevos generalmente utilizan Eclipse porque provee la infraestructura básica para la creación del IDE requerido. Para comenzar, descargaremos la versión para **Java Developers**. El archivo posee una extensión .ZIP (archivo comprimido), que descomprimiremos en el lugar que deseemos. Una vez descomprimida la carpeta, ejecutaremos el archivo eclipse.exe (es conveniente crear un acceso directo desde el escritorio para futuros usos).

Lo primero que veremos será una ventana que nos preguntará dónde queremos guardar el workspace o área de trabajo, un directorio donde Eclipse salvará los distintos proyectos en los que trabajemos. Luego veremos la pantalla de bienvenida con ejemplos y tutoriales, así como también una visita a las características de la herramienta. Si elegimos ir al workbench (espacio de trabajo), se presentará con la vista Java. Cada **vista** es una configuración de paneles y ventanas que le permiten al usuario enfocarse en una determinada tarea. Además de la vista Java, encontraremos las vistas Java Browsing y Debug.

Figura 4. La opción vista Java es la principal de Eclipse. El Package Explorer, el área de edición y el área de problemas son los sectores más importantes.

Si queremos aprender más sobre cómo trabajar con Eclipse IDE, podemos buscar en **www.eclipsetutorial.sourceforge.net** y ver la primera lección del tutorial para principiantes (Total beginners). Nos enteraremos de muchas funcionalidades de Eclipse, que agilizarán el desarrollo de nuestros sistemas.

La vista Java posee un explorador de paquetes (Pakage Explorer), fuentes Java y librerías del proyecto. Incluye también el panel de errores (Problems), donde Eclipse informará aquellos relacionados con el proyecto (librerías que faltan, errores sintácticos en los códigos fuente, etcétera). El panel del esquema (Outline) de las clases y paquetes también está presente, y la parte central de la vista está destinada a la edición de código.

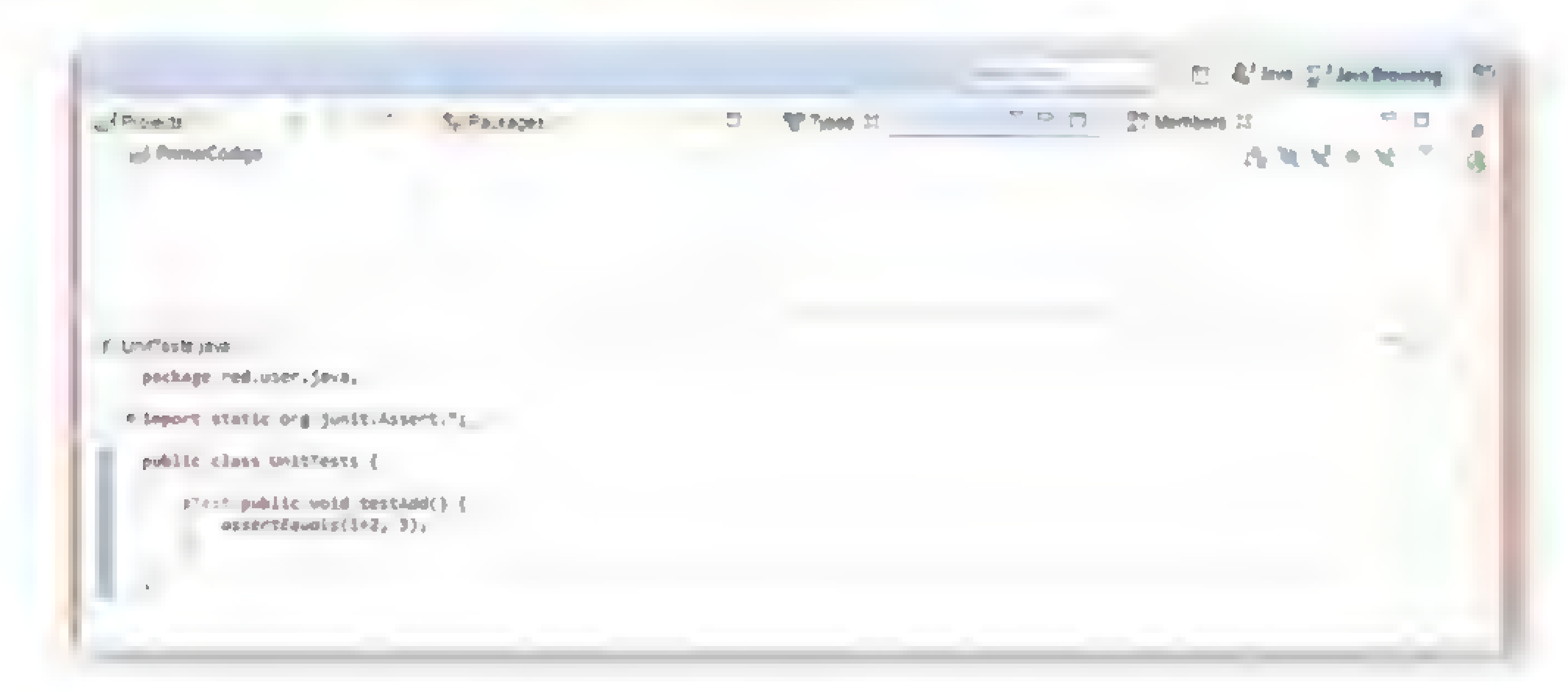
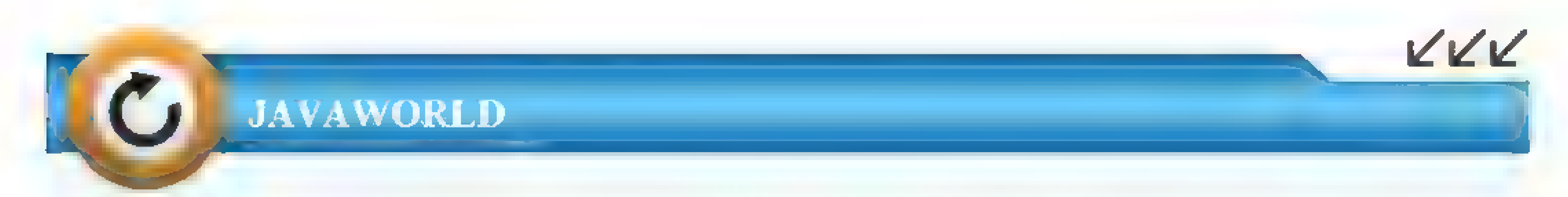


Figura 5. La vista Java Browsing focaliza la atención en el área de edición y en la navegación de paquetes, clases y métodos.

La segunda vista (Java Browsing) está enfocada en el trabajo con las clases y nos recuerda al entorno de trabajo clásico de **Smalltalk**. En esta vista nos encontramos, en la parte superior, con paneles, mientras que el resto del espacio es destinado a la edición de código fuente.



En **www.javaworld.com** encontraremos abundante información sobre este lenguaje. Allí hallaremos las últimas noticias, artículos, tutoriales y enlaces a distintos blogs que nos serán de gran utilidad y nos mantendrán actualizados sobre los temas claves del ambiente Java.

Los cuatro paneles son: el de proyecto (Proyects), el de paquete (Packages, que muestra los distintos paquetes que hay), el de clases (Types, que presenta las distintas clases dentro del paquete seleccionado en el panel anterior) y el de miembros (Members, que muestra los elementos que conforman la clase seleccionada en el panel anterior).

Figura 6. La vista Debug permite que seamos testigos de lo que ocurre durante la ejecución del código y ayuda a encontrar errores de una manera sencilla.

Finalmente, la vista Debug se activa cuando la ejecución (lanzada en modo Debug) alcanza un **breakpoint** (punto de frenado) en el código. En ella podemos hacer avanzar la ejecución, detenerla, inspeccionar qué objetos están siendo usados en un determinado momento e incluso modificarlos. Esta es la vista que más usaremos cuando necesitemos determinar exactamente qué está haciendo el código y por qué no hace lo que le estamos indicando.

Test Driven Development

A lo largo del libro utilizaremos, a través de Eclipse, **JUnit**, una herramienta para trabajar con la metodología **TDD** (*Test Driven Development* o desarrollo guiado por pruebas). TDD es un estilo para plantear el desarrollo de un software, en el cual el programador vuelca su conocimiento acerca de un determinado problema en forma de aserciones.

Las aserciones permiten evaluar si nuestra aplicación se encuentra en los valores esperados y cumple con lo especificado. Son un modo de comprobar que esté funcionando correctamente una línea de código. Por ejemplo:

```
assert ExpresionBooleana [: Expresion_que_expresa_el_error];
```

El programador debe codificar lo necesario para validar estas aserciones. Cuando todas sean válidas, debe seguir explorando el dominio del problema agregando más aserciones. Cuando alguna resulte falsa, debe codificar para hacerla verdadera. Finalmente, el programador refactoriza el código y lo repara en búsqueda de abstracciones, a fin de simplificarlo sin que cambie el comportamiento; las aserciones deben continuar siendo válidas.

Abordar el problema desde TDD es más fácil que tratar de resolverlo en abstracto y para todos los casos. Asimismo, TDD hace que el programador sea usuario de su propio código y pueda darse cuenta de cuán amigable y fácil de usar es la API que está desarrollando.

Por otra parte, TDD también facilita el mantenimiento del software una vez que madura. El software, por su naturaleza, está en constante cambio, y estos cambios tienen que ser codificados e introducidos en el código existente sin afectar involuntariamente al resto de las funcionalidades del sistema. Así, los tests sirven como red de seguridad, dado que podremos verificar rápida y sistemáticamente el correcto funcionamiento de la aplicación.

BDD O BEHAVIOR DRIVEN DEVELOPMENT

BDD (desarrollo guiado por comportamiento) es una extensión de TDD que hace foco en escribir los tests en un lenguaje cercano al natural, con las especificaciones obtenidas del cliente y las que este puede entender. De esta forma, la brecha entre el programador y el cliente se reduce, ya que comparten un mismo léxico.

JUnit es la herramienta clásica, originaria de Smalltalk, para realizar TDD en Java (otra conocida es **TestNG**). Su aparición se dio de la mano de Kent Beck, creador del **Extreme Programming** (programación extrema), y luego fue migrada a Java y a casi todos los lenguajes.

> JUNIT ES UNA HERRAMIENTA DE SMALLTALK PARA REALIZAR TDD EN JAVA

En JUnit se definen casos de prueba, representados por una clase que contiene métodos de prueba. Cada uno de estos métodos tiene el objetivo de probar ejemplos y aserciones. JUnit ejecuta los casos y genera un reporte que informa cuáles pasaron exitosamente y cuáles fallaron o dieron error. En Eclipse contamos con un plugin ya instalado para utilizar JUnit con unos pocos clics.

Primeros códigos

Para trabajar en nuestros primeros códigos en Java, deberemos crear, primero, un proyecto donde escribirlos. Iniciaremos Eclipse y, posteriormente, haremos clic en el menú File o sobre el botón New, que se encuentra en la barra de herramientas (también podemos hacer clic con el botón derecho del mouse en el Package Explorer y seleccionar New.../Java Project). Elegiremos un nombre para el proyecto y, luego, presionaremos Finish.

EXTREME PROGRAMMING

Se trata de una metodología ágil de desarrollo de software creada por Kent Beck a mediados de los noventa. Se la conoce como un conjunto de prácticas tales como **pair programming** (programación de a pares), **code reviews** (revisar el código de otros), **unit testing**, alta comunicación en el equipo, tener al cliente dentro del equipo y desarrollo iterativo.

Figura 7.
Pantalla de creación de proyectos Java. A través de las opciones que ofrece el asistente, podemos controlar cada parámetro de la creación.

Con esto conseguiremos tener un proyecto y dentro de la carpeta src encontraremos alojados los archivos fuente. De modo predeterminado, Eclipse agrega como dependencia las librerías que vienen con la instalación de Java.

El paso siguiente es crear nuestro primer caso de prueba (Test Case). En Java, todo código tiene que estar contenido en una clase, y nuestro Test Case no será la excepción. Para crearlo, en el menú File seleccionaremos New.../JUnit Test Case , elegiremos una versión de JUnit, un nombre de package (red.user.java) y un nombre para el Test Case (UnitTests). Luego, presionaremos Finish y agregaremos la librería con OK.

Haremos ahora una prueba simple. Veamos si, efectivamente, cuando sumamos uno más dos, obtenemos tres. Para esto, escribamos el código que aparece en la página siguiente:

Figura 8.
La pantalla de creación del Test Case nos permite elegir la versión de JUnit con la que queremos trabajar y crear métodos auxiliares para los tests.

```
package red.user.java;

import org.junit.Test;
import static org.junit.Assert.*;

public class UnitTests {

	@Test public void testAdd() {
			assertEquals(1+2, 3);
	}

}
```

Esta es la forma de un Test Case, una clase que contiene los métodos que comprueban si el código funciona. Veamos algunos de los elementos que componen este código para entenderlo mejor.

Figura 9. Así es como luce nuestro código en Eclipse. Notemos los estilos que utiliza el IDE para diferenciar los elementos que conforman el código.

La palabra class, seguida del nombre de la clase, indica el inicio. Luego tenemos el método (testad), que no devuelve ninguna respuesta (void o vacío) y que está marcado como un método de test (@Test).

En el cuerpo del método, encontramos una aserción que dice: "¿Son iguales 3 y 1+2?". Para saber si es así, ejecutaremos el test y veremos qué resulta. Presionaremos el botón derecho sobre la clase y seleccionaremos Run As.../ JUnit Test. Aparecerá el **runner** de JUnit, que mostrará una barra verde si todo salió bien, o roja, si algo falló.

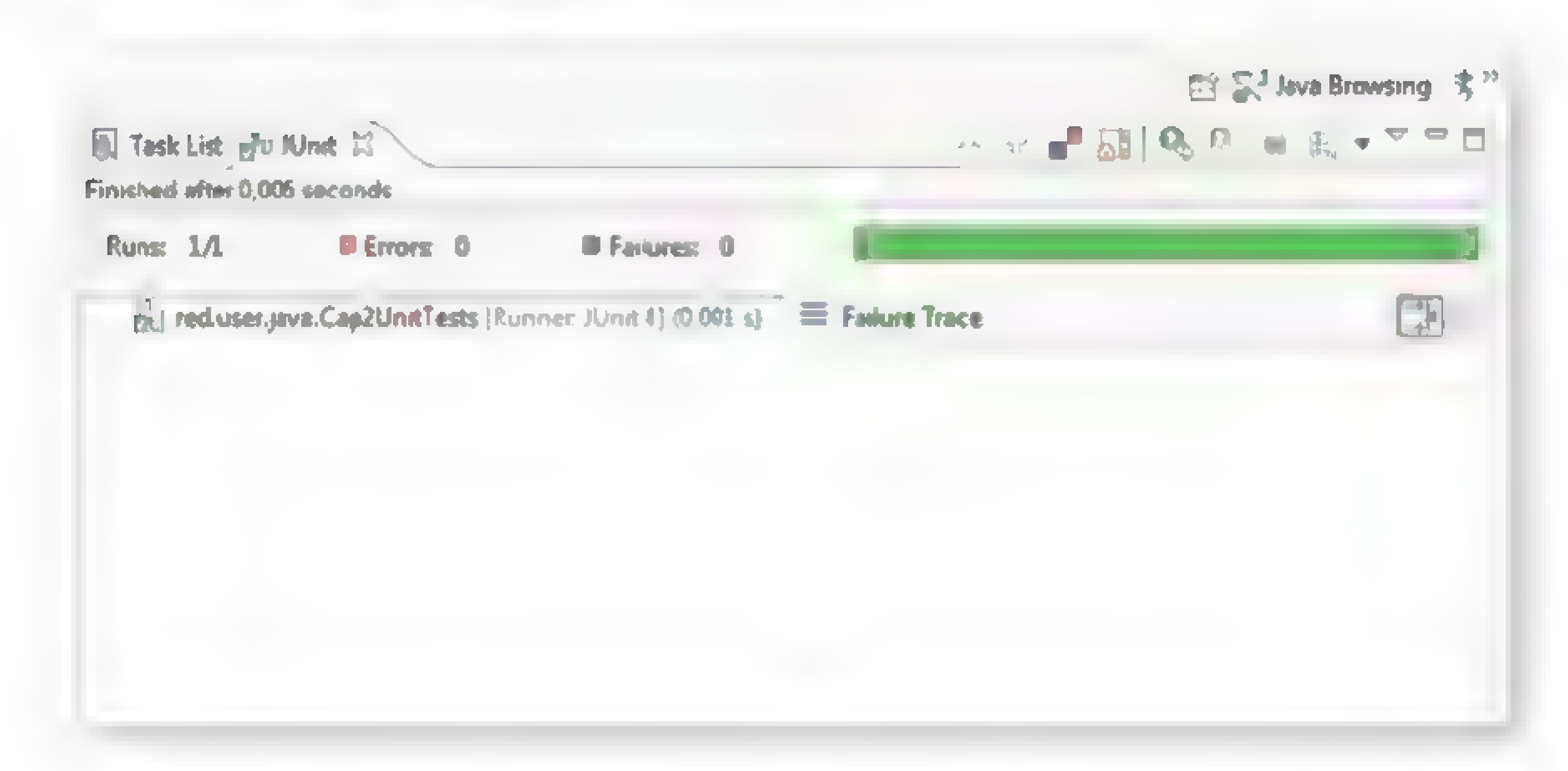


Figura 10. Al correr un Test Case, el resultado se ve reflejado en el panel del **runner** de JUnit. Si la barra es verde, todos los tests pasaron; si es roja, alguno falló.

En el sitio **www.junit.org** podemos encontrar más información sobre JUnit, con documentación y ejemplos que nos ayudarán a mejorar la forma en que hacemos los tests. La sección de preguntas y respuestas es un buen punto de partida para resolver todas las dudas que tengamos acerca de estos. Además, podemos bajar el código fuente y aprender del funcionamiento interno de JUnit.

Ahora podemos investigar con más profundidad qué hemos escrito y cómo es que funciona:

```
package red.user.java;
```

Todo archivo fuente Java define una **clase**, y toda clase pertenece a un **paquete**. Un paquete es una forma de agrupar las clases en grupos que tengan cierto sentido. Se estila que el prefijo del paquete sea el dominio web de la organización a la que pertenece, pero en orden inverso (por ejemplo, **com.google**).

En nuestro código, definimos el paquete red; dentro de él, el subpaquete user y, dentro de este, el subpaquete java. Java fuerza a que cada paquete esté representado por un directorio en el sistema de archivos. En nuestro caso, Eclipse automáticamente crea un directorio red; dentro de él, otro llamado user; adentro, uno llamado java; y, finalmente, dentro de java, el archivo UnitTests.java.

```
import org.junit.Test;
import static org.junit.Assert.*;
```

ANOTACIONES

Las anotaciones son un artefacto de Java que puede ser utilizado para agregar información acerca de los distintos elementos del lenguaje. Esta información es llamada **metadata**, una palabra que significa "datos sobre los datos", es decir, datos que dicen algo respecto de otros datos. El prefijo **meta** conlleva el significado de "hablar en un nivel superior".

Estas sentencias indican que queremos importar la clase Test, localizada en el paquete org.junit, y también todos los **métodos estáticos** (que pertenecen a la clase y no a las instancias de ella) de la clase Assert. En Java, los métodos estáticos son distintos de los de **instancias**.

```
public class UnitTests
```

Hemos indicado que estamos definiendo la clase UnitTests. Todo lo que esté entre llaves corresponde con el cuerpo del código en la clase { }.

```
@Test public void testAdd()
```

Aquí definimos el método que ejecutarán las instancias de la clase UnitTests cuando se les envíe el mensaje testAdd. Primero tenemos la anotación @Test, que indica que el método es un test. Aunque la anotación dice que el método es un test, no dice nada acerca de lo que hace; de eso habla el método en sí.

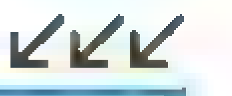
Luego, el método está marcado como **público**, lo que significa que puede ser invocado (es decir, que se les puede mandar el mensaje a las instancias) desde instancias de otras clases. Como dijimos, el método no devuelve ninguna respuesta, de ahí que esté marcado como void. Al igual que en la clase, el cuerpo del método está demarcado con las llaves.

PARA DESARROLLADORES JAVA

En el sitio **www.oracle.com/technetwork/java/index.html** podremos descargar el kit de desarrollo de la versión estándar de Java, **Java SE**. También encontraremos los kits de desarrollo más avanzados, para aplicaciones empresariales (**Java EE**), de celulares (**Java ME**) y para aplicaciones gráficas (**Java FX**).

```
assertEquals(1+2, 3);
```

Finalmente, tenemos la aserción. En ella encontramos la invocación al método estático, perteneciente a la clase Assert (recordemos el import static), assertEquals. Este método recibe dos parámetros que compara para saber si son iguales o no. En este caso, recibe el resultado de evaluar 1+2 (en Java esto no es un envío de mensaje, sino que lo ejecuta directamente la máquina virtual) y el número 3.



RESUMEN

A través de este capítulo pudimos conocer la historia de Java y algunas circunstancias que rodearon su aparición. Conocimos los alcances de este lenguaje y vislumbramos todo su potencial. Luego analizamos las características del entorno de desarrollo con la integración de Eclipse y creamos una primera y sencilla aplicación.

Actividades

TEST DE AUTOEVALUACIÓN

1 ¿Qué es una **máquina virtual**?

2 ¿Por qué triunfó Java frente a otras tecnologías?

3 ¿Cuál es la diferencia entre **JRE** y **SDK**?

4 ¿Qué es y para qué sirve **Eclipse**?

5 ¿Qué es **TDD**?

EJERCICIOS PRÁCTICOS

1 Investigue qué errores nos muestra Eclipse (en el panel Problems, al salvar el código fuente) al borrar distintas partes del código.

2 Trate de utilizar el compilador del SDK para el test realizado.

3 Pruebe ejecutar el resultado de la compilación con el comando java (dará error).

4 Haga doble clic sobre el costado gris izquierdo del método: aparecerá un círculo azul, que indica un **breakpoint**. Ejecute el Test Case en modo Debug, eligiendo Debug... en vez de Run....

PROFESOR EN LÍNEA

Si tiene alguna consulta técnica relacionada con el contenido, puede contactarse con nuestros expertos: **profesor@redusers.com**.

Sintaxis

La sintaxis es el conjunto de reglas que dan como resultado un programa adecuadamente desarrollado. Estas reglas describen qué elementos están permitidos en determinados contextos y cuál es la relación entre ellos. En este capítulo, conoceremos la sintaxis básica necesaria para dominar la programación en Java.

Palabras clave

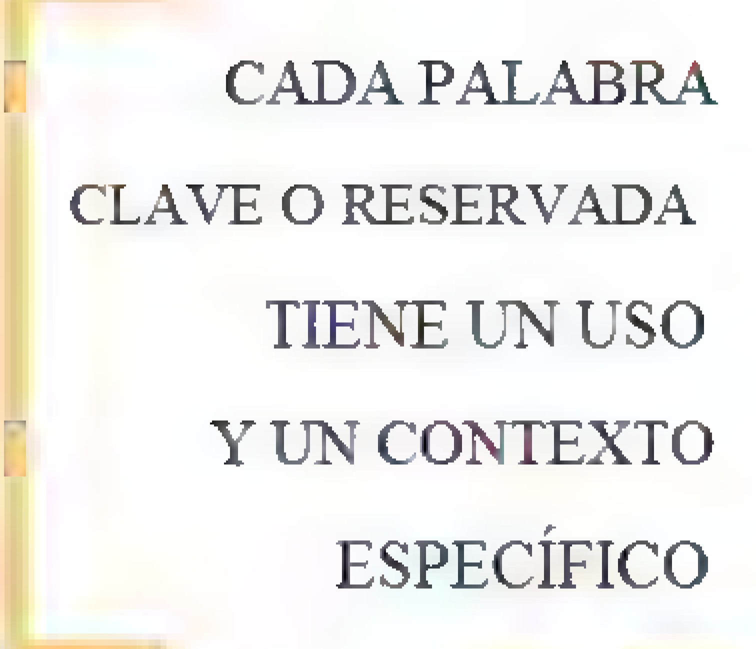
Las **palabras clave** o reservadas son palabras que Java no permite que los programadores utilicen libremente en cualquier parte del código (por ejemplo, los nombres de variables), sino que tienen un propósito determinado y un contexto específico. Para indicar que una palabra está reservada, Eclipse la resaltará con un color distinto del texto normal. A continuación, encontraremos el listado de palabras clave, su explicación y uso.

- abstract: se utiliza en clases y métodos para indicar que son abstractos. En las clases, significa que sirven para formar una jerarquía y no pueden tener instancias. En los métodos, para indicar que la jerarquía que tienen responde al mensaje, pero no define código alguno.
- assert: se utiliza para enunciar condiciones que se deben cumplir durante la ejecución. No debe confundirse con las aserciones de JUnit.
- boolean: representa al tipo primitivo booleano. Las variables de este tipo no son objetos, y los únicos valores que pueden tomar son true (verdadero) y false (falso).
- break: es una etiqueta que se utiliza para salir de la ejecución de los ciclos (for, while, do while) y de las estructuras switch.
- byte: representa el tipo primitivo de números de 8 bits, entre -128 y 127. Los valores de este tipo no son objetos.

UNICODE

Carácter es la representación de un símbolo, ya sea una letra, número o símbolo. La referencia **Unicode** especifica un nombre e identificador numérico único para cada carácter o símbolo diseñado, para facilitar el tratamiento informático de múltiples lenguajes y disciplinas técnicas, además de textos clásicos de lenguas muertas. El término proviene de las palabras **universalidad**, **uniformidad** y **unicidad**

- case: etiqueta usada en las estructuras switch para indicar un caso.
- catch: se utiliza en conjunto con la etiqueta try para indicar que se espera una excepción en el código envuelto en la estructura de try.
- char: representa el tipo primitivo de los caracteres. Las variables de este tipo no son objetos y pueden contener cualquier **carácter Unicode**.
- class: define el comienzo de la declaración de una clase.



- continue: al igual que break, es un etiqueta que se utiliza en los ciclos, y sirve para hacer que se interrumpa la ejecución del código de la iteración actual del ciclo y que continúe con la siguiente.
- default: se usa para definir la acción por defecto en las estructuras switch.
- do: este término se utiliza en combinación con el while para formar la estructura de ciclo do while.
- double: representa el tipo primitivo de los números de punto flotante de doble precisión (64 bits) definidos en el estándar **IEEE 754**.
- else: representa el camino alternativo en un toma de decisión; se utiliza en conjunto con el if.
- enum: declara un tipo enumerado (clases que tienen un número fijo y definido de instancias nombradas).
- extends: se utiliza para indicar que una clase hereda de otra. Si no se utiliza, por defecto toda clase hereda de la clase Object. También se usa para declarar que una interfaz extiende otra (**herencia entre interfaces**).
- final: es un modificador que puede aplicar a clases, métodos variables y argumentos, y que puede tener un significado distinto en cada caso. En las clases, significa que se puede extender heredando. En el caso de los métodos, que las subclases de la clase que lo contiene no pueden redefinirlo. En las variables (tanto las de instancia como las de clase, así como en las definidas en el cuerpo de los métodos), significa que se puede asignar valor solo una vez.

Finalmente, en los argumentos de los métodos, quiere decir que esa variable no se puede escribir (o sea, reasignarle un valor nuevo).

- finally: se usa en conjunto con la estructura try, y especifica un bloque de código que se ejecuta, sí o sí, al finalizar la ejecución del bloque try.
- float: representa al tipo primitivo de los números de punto flotante de simple precisión (32 bits) definidos en el estándar **IEEE 754**.
- for: es la etiqueta que define la estructura de ciclo más usada.

LA ETIQUETA FOR DEFINE LA ESTRUCTURA DE CICLO MÁS USADA

- if: especifica una bifurcación en la ejecución del código guiada por una condición. Si la condición es verdadera (true), entonces se ejecuta el cuerpo del if; si no, se lo saltea.
- implements: es la manera de indicar que una clase implementa (hereda) de una o varias interfaces. La clase está obligada a implementar los métodos definidos por las interfaces que utiliza (a menos que sea una clase abstracta, en cuyo caso las encargadas serán las subclases).
- import: declara el uso (importación) de clases y paquetes en un código fuente. Mediante el agregado del modificador static, también se pueden importar métodos estáticos de una clase.
- instanceof : instrucción que sirve para preguntar si un objeto es instancia de una determinada clase (o superclase) o interfaz. El resultado es true o false.
- int: representa el tipo primitivo de números de 32 bits, entre -2147483648 y 2147483647. Los valores de este tipo no son objetos.
- interface: especifica el comienzo de la definición de una interfaz.
- long: representa el tipo primitivo de números de 64 bits, entre -9223372036854775808 y 9223372036854775807. Los valores de este tipo no son objetos.
- native: modificador de método que indica que este no está implementado en Java sino en otro lenguaje (generalmente, en C++).

Se utiliza para hacer que Java se comunique con el exterior de la máquina virtual (por ejemplo, para acceder al sistema de archivos o a la red).

- new: es el operador de creación de instancias. Se utiliza junto con el nombre de la clase no abstracta que se quiere incluir en la creación de la instancia y los parámetros necesarios para la inicialización de esta. Como resultado, se obtiene una nueva instancia de la clase.
- package: instrucción que define a qué paquete pertenece la clase. En Java, la estructura de paquetes tiene que coincidir con la estructura de directorios.
- private: modificador que aplica a clases, métodos y atributos (de instancia y de clase). Indica cuál es el nivel de acceso al elemento. Aplicado a una clase (solamente para clases definidas dentro de clases) indica que solo se la puede utilizar dentro de la clase que la definió. En métodos, indica que solo se los puede invocar desde otros métodos de la clase que los define. En los atributos, significa que no pueden ser accedidos, salvo por los métodos definidos en la clase que los define. Tanto en el caso de los métodos como en el de los atributos, no pueden ser accedidos por otras clases, ni siquiera por subclases de la clase que los contiene.
- protected: al igual que private, es un modificador de acceso, y aplica a los mismos elementos, con la diferencia de que permite que los elementos sean también accedidos por subclases.
- public: es un modificador de acceso que indica que el elemento en cuestión es público y puede ser accedido por cualquier código. Aplica a clases, métodos y atributos.

ESPACIOS EN BLANCO

Java utiliza cualquier espacio en blanco (espacios, saltos de líneas y sangrías, uno o más de los anteriores) para separar las palabras del código. Son necesarios a menos que haya algún otro símbolo que separe (paréntesis, comas, llaves, etcétera).

- return: es la instrucción que indica que se termina la ejecución de un método y, opcionalmente, devuelve un valor como respuesta.
- short: representa el tipo primitivo de números de 16 bits, entre -32768 y 32767. Los valores de este tipo no son objetos.
- static: tiene tres usos bien distintos. El primero y más importante es indicar que un método o un atributo pertenece a la clase en vez de ser de instancia (se dice que son **estáticos**). El segundo es especificar la importación de los métodos estáticos de una clase, en conjunto con import. Finalmente, el tercer y último uso se da al declarar clases e interfaces dentro de otra clase (o interfaz). La diferencia entre indicar que sea estática o no la clase (o interfaz) interna es que puede ser accedida como cualquier elemento estático, solo utilizando el nombre de la clase contenedora. De otra forma, se requiere de una instancia para tener acceso a la clase.
- strictfp: antes de la versión 1.2 de la **Java Virtual Machine**, todas las operaciones con punto flotante daban como resultado valores estrictos respecto del estándar **IEEE 754**. Pero, a partir de esa versión, Java utiliza para los cálculos intermedios otras representaciones que puedan estar disponibles en la máquina. De esta manera, se obtiene mayor precisión en las operaciones, pero los resultados no son transportables. Para forzarlos a que sí lo sean, se puede utilizar este modificador de método.
- super: es un identificador relacionado con this, dado que también es una referencia al objeto actual (receptor del mensaje que se está ejecutando), pero con la diferencia de que se utiliza para indicar que se quiere ejecutar un método que se encuentra en alguna de las superclases de la clase del método que se está ejecutando. Es una marca para modificar el comportamiento de method lookup, para que no comience la búsqueda del método en la clase propia del objeto receptor. También se utiliza para especificar un cota inferior a los tipos de un genérico.
- switch: sirve para declarar una estructura switch. Dicha estructura se utiliza para comparar una variable de tipo primitivo con ciertos valores y, en caso de que coincida con alguno, ejecutar cierto código. Se utiliza con las etiquetas case, break y default.

- synchronized: es tanto un modificador como una instrucción. Como modificador, aplica solo a los métodos de instancia y tiene como objetivo sincronizar el acceso concurrente de distintos códigos ejecutándose en paralelo. Lo que hace es forzar que solo un código (**threads** o hilos de ejecución, son procesos livianos) pueda estar ejecutando métodos sincronizados del objeto. Los threads son frenados y esperan a que el thread en ejecución termine, y así le toca al siguiente. La forma de instrucción sirve para lo mismo, pero se utiliza dentro del cuerpo de los métodos y delimita la zona de código que quiere sincronizar. Además, requiere que se especifique cuál es el objeto que se va a utilizar para realizar la sincronización (en el caso del modificador, el objeto utilizado era el receptor del mensaje).

LA ESTRUCTURA SWITCH SE UTILIZA CON LAS ETIQUETAS CASE, BREAK Y DEFAULT

- this: es una referencia a la instancia actual que está ejecutando el método en cuestión. Es la instancia receptora del mensaje que corresponde con la ejecución del método. Solamente se puede utilizar en el cuerpo de los métodos de instancia.
- throw: es la instrucción utilizada cuando se debe lanzar una excepción durante la ejecución.
- throws: indicador de que un método lanza excepciones. El compilador se asegura de que el emisor del mensaje que origina una excepción de estas la atrape con la instrucción catch o que indique que también la lanza (deja pasar) al anotarse en conjunto con throws y el nombre de la clase de la excepción.
- transient: es un modificador de atributos que los marca como no serializables. Significa que, cuando un objeto es convertido a bytes para ser transmitido por red o a un archivo, los atributos marcados como transient son ignorados. En el proceso de reconstrucción, estos son inicializados a su valor por defecto.

- try: etiqueta utilizada para delimitar un bloque de código. Se utiliza en conjunto con las etiquetas catch y/o finally.
- void: indica que un método no devuelve nada como respuesta.
- volatile: es un modificador de atributos (tanto de instancia como de clase) que indica que el atributo es accedido y modificado por varios threads. Si bien todo atributo puede ser accedido concurrentemente, la JVM toma recaudos con estos atributos forzando que las operaciones de escritura y de lectura de estos sean atómicas.
- while: etiqueta usada para los ciclos while y do while.

Java posee un par de palabras reservadas que, aunque no se utilizan, se encuentran en la definición del lenguaje. Ellas son const y goto, resabios de versiones primitivas del lenguaje.

Asimismo, se definen también las siguientes palabras clave, que son objetos denominados **literales**.

- false: es el elemento que representa la falsedad, su tipo es boolean.
- true: es el elemento que representa la verdad, su tipo es boolean.
- null: representa el vacio, la nada. Solo las referencias a objetos pueden estar asignadas a null (los valores primitivos, no), que no tiene tipo (no es instancia de ninguna clase) pero puede ser asignado a cualquier referencia, o pasado como parámetro cuando se espera un objeto.

Ciclos

Los **ciclos** son estructuras de código que nos permiten representar situaciones similares a las siguientes: "mandar a cada cliente un e-mail con las ofertas del mes" o "mientras haya papas en el cajón, tengo que pelarlas".

En Java tenemos cuatro estructuras de ciclos, todas con el mismo poder, pero cada una con una expresividad distinta. Tenemos el ciclo for, el while, el do while y el for each.

El ciclo for

El ciclo for ('para') es una de las estructuras más usadas en los lenguajes de programación. Consta de cuatro partes: la **inicialización**, la **condición**, el **paso siguiente** y el **cuerpo del ciclo**. Veamos esto en Java:

```
int sum = 0;
for(int i = 0; i < 4; i++) {
        sum += i;
}
```

El fragmento de código anterior suma los números 0, 1, 2 y 3 en la variable sum. La primera línea es la declaración e inicialización de la variable sum en 0. Luego tenemos el ciclo for, donde podemos apreciar la inicialización (int i = 0), donde declaramos otra variable (que solo puede ser accedida por el código for). Separada por un punto y coma tenemos la condición (i < 4). Después, también separado por un punto y coma, encontramos el paso siguiente (i++), que en este caso dice "incrementar en 1 la variable i".

El cuerpo del for es el código comprendido entre las llaves. En él encontramos la sentencia sum += i;, que es igual a escribir sum = sum + i; (o sea, sumar lo que hay en sum con lo que hay en i y guardar el resultado en sum).

La semántica de for es la siguiente: "para i igual a 0, mientras i sea menor que 4, sumar sum e i y guardarlo en sum; luego incrementar i en 1 y repetir".

ESTRUCTURAS VERSUS MENSAJES

Las estructuras como el if o el for están en la sintaxis del lenguaje y no representan ningún envío de mensajes. Esto hace que sean un elemento extraño al paradigma. Algunos lenguajes, como **Smalltalk**, tienen el mismo comportamiento al enviar mensajes; esto los hace más flexibles.

Formalmente, la estructura del for acepta cualquier sentencia válida de Java –e incluso nada– como inicialización y se ejecuta solamente una vez al inicio del for. Acepta cualquier sentencia que dé como resultado un valor boolean como condición (nada significa true), que se pregunta al inicio de cada iteración (paso o ciclo), y ejecuta el código si el resultado es true e interrumpe el for si es false. Finalmente, cualquier expresión valida para el paso siguiente, que se ejecuta al final de cada ejecución del código del cuerpo.

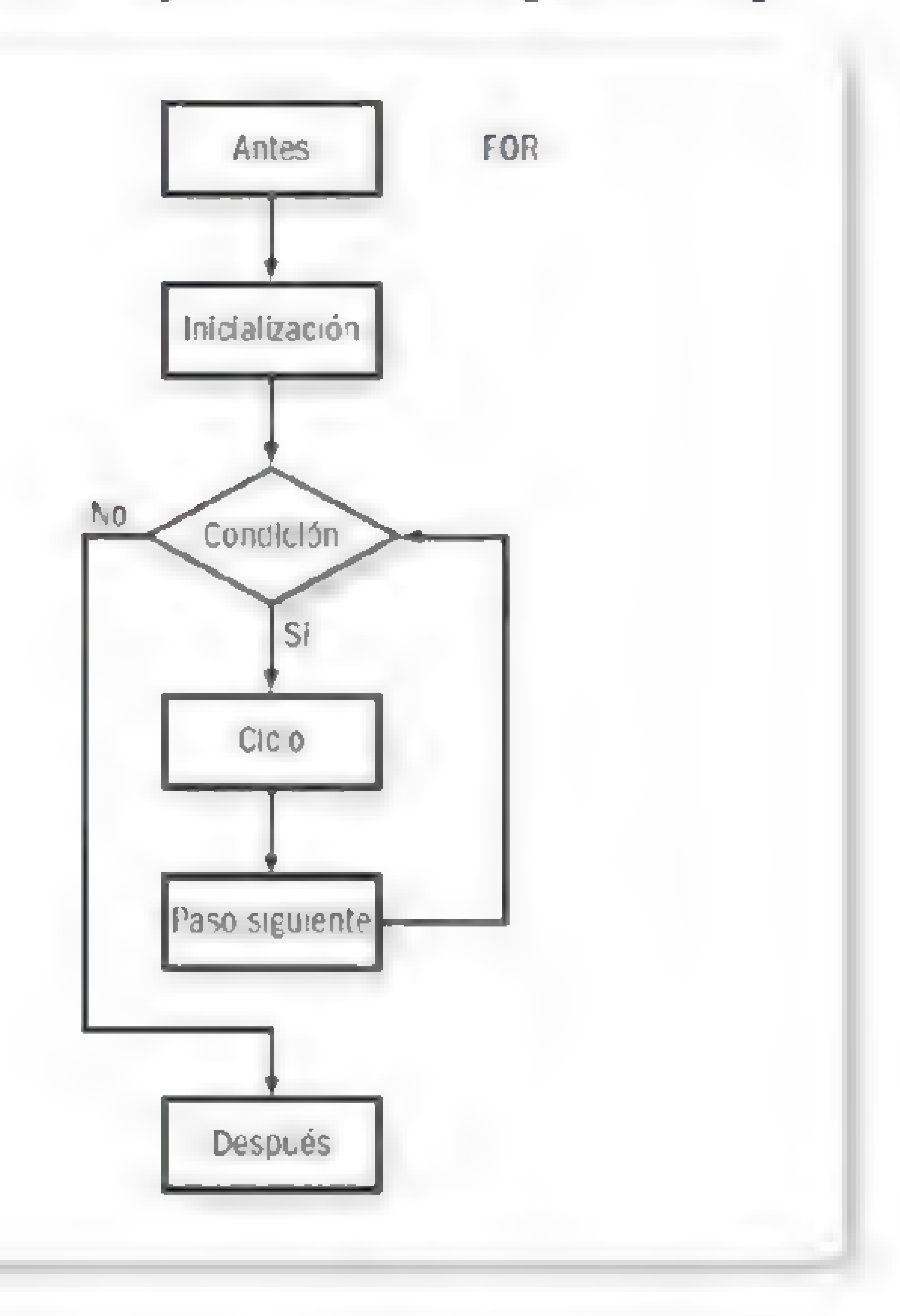


Figura 1. Diagrama de flujo correspondiente al ciclo for.

Dentro del cuerpo del ciclo es posible utilizar las etiquetas break y continue para alterar su comportamiento. La etiqueta break se usa para abortar la ejecución del for completamente, mientras que continue se utiliza para terminar la ejecución de la iteración actual y ejecutar la siguiente.

Veamos un ejemplo:

```
int sum = 0;
for(int i = 0; i < 4; i++) {
        if(i == 1) continue;  // Si i es igual a 1 ir al paso siguiente
if(i == 3) break; // Si i es igual a 3 abortar el for
        sum += i;
}
assertEquals(sum, 2);
```

El ciclo while

El ciclo while es más simple que for, dado que solamente tiene la **condición** y el **cuerpo**. Un ciclo while que sume los números del 0 al 3 se ve del siguiente modo:

```
int sum = 0;
int i = 0;
while(i < 4) {
        sum += i;
        i++;
}
```

Como vemos, comparando este código con el del ciclo for, tenemos la inicialización fuera de la estructura de while y el incremento de la variable i dentro del cuerpo.

A while hay que leerlo de esta forma: mientras se cumpla la condición, ejecutar el cuerpo. Al igual que en el for, y en todas las otras estructuras de ciclos, podemos utilizar las etiquetas break y continue.

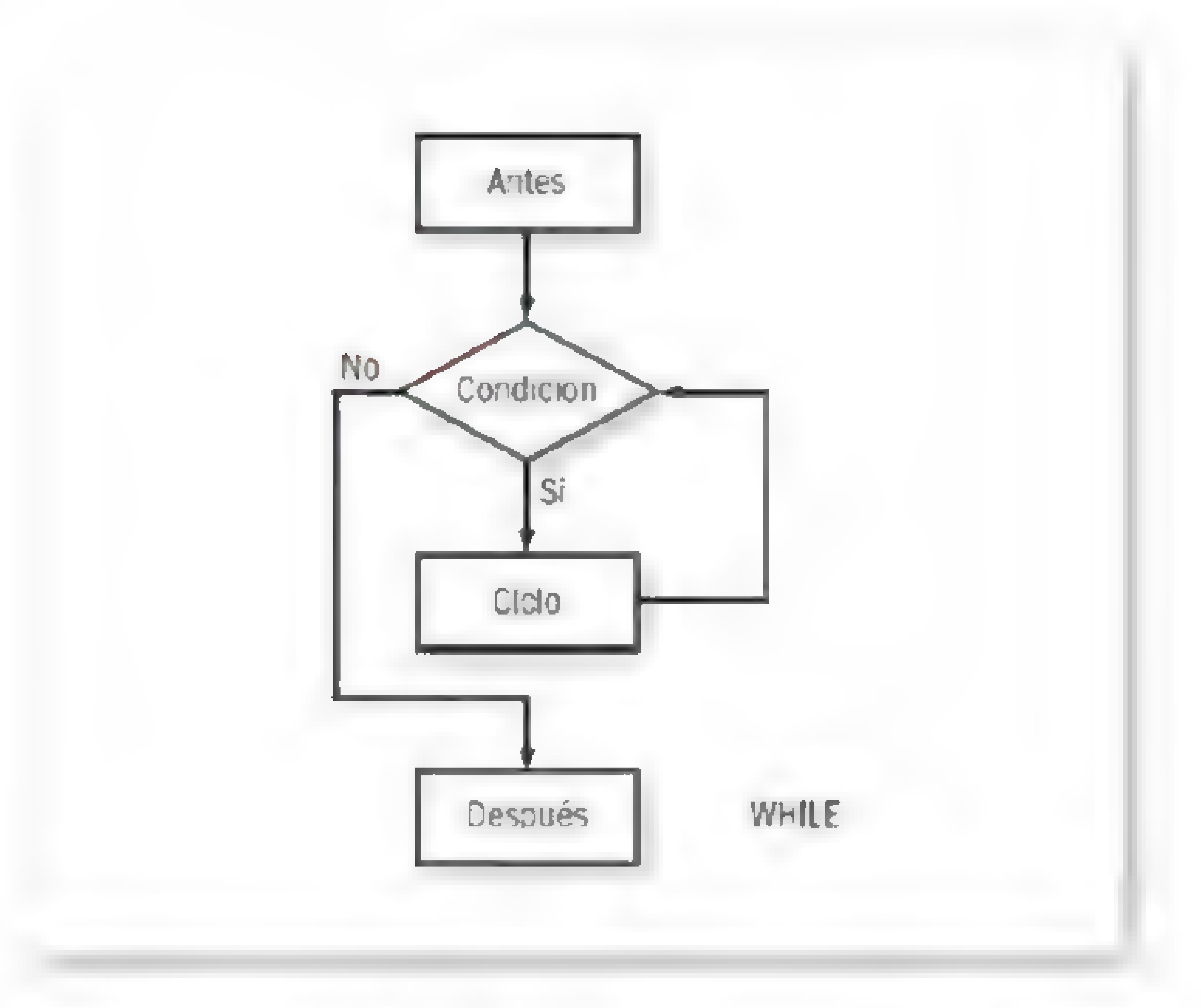


Figura 2. En la figura se observa un diagrama de flujo correspondiente al ciclo while.

El ciclo do while

El do while es una variación del while, donde primero se ejecuta el cuerpo y luego se pregunta si se cumple la condición para continuar iterando o no.

```
int sum = 0;
int i = 0;
do {
        sum += i;
        i++;
} while(i < 4);
```

Hay que ser cuidadosos a la hora de usar esta estructura, porque el cuerpo se ejecuta si o si al menos una vez.

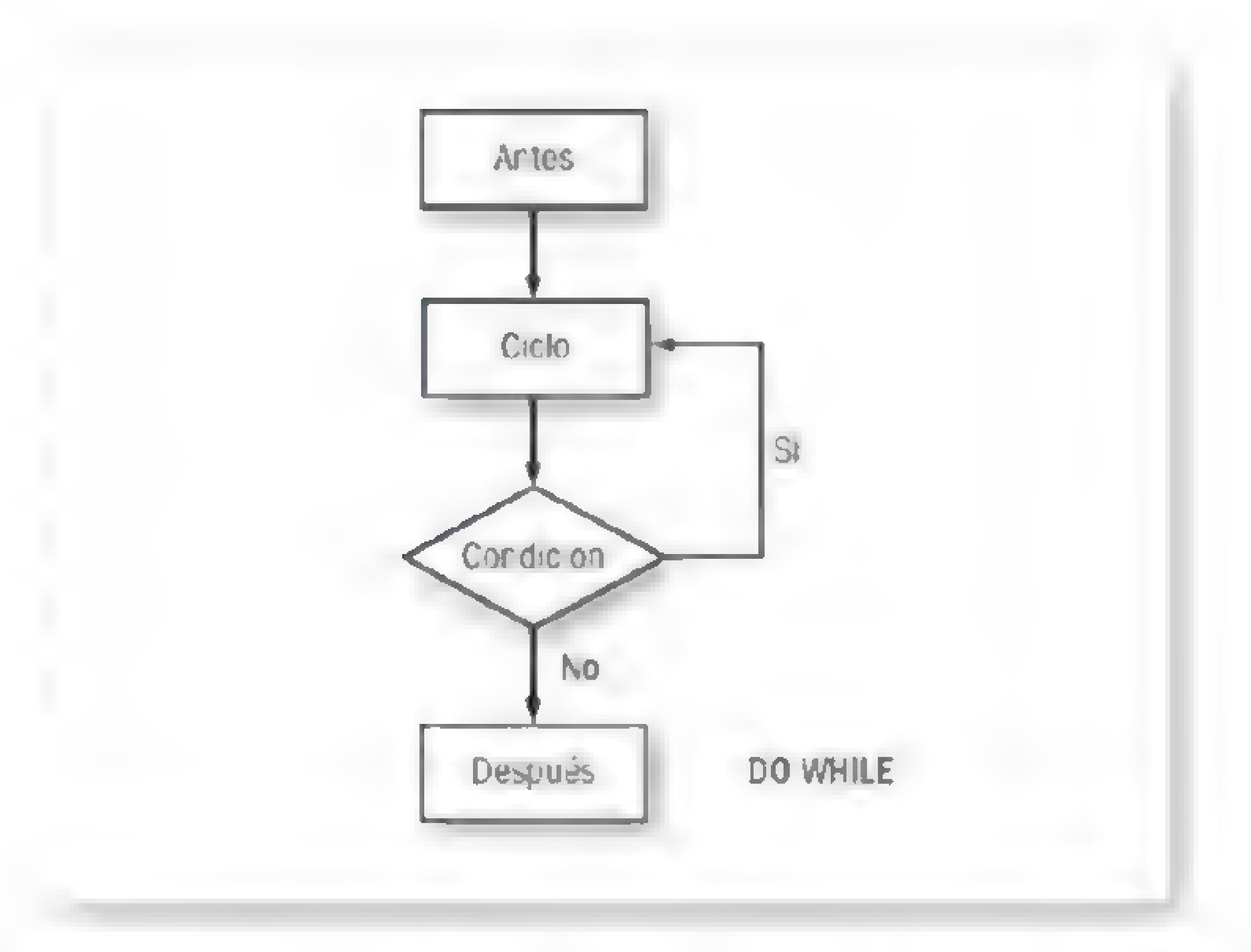


Figura 3. Diagrama de flujo correspondiente al ciclo do while.

El ciclo for each

Este ciclo sirve para recorrer estructuras tales como listas y otras colecciones de objetos. Estas estructuras se conocen como **iterables** y responden al mensaje iterator, que devuelve una instancia de iterator (un objeto que sabe cómo recorrer la estructura). El for each es una variación del for para ser usada con estas estructuras iterables de una forma más práctica y simple.

```
// Asumamos que "dígitos" es una colección con los números del 0 al 9
...
int sum = 0;
for(int digito : digitos) {
        sum += digito;
}
assertEquals(sum, 45);
```

Leemos el código de esta forma: por cada digito en digitos, sumar sum con digito y guardarlo en sum.

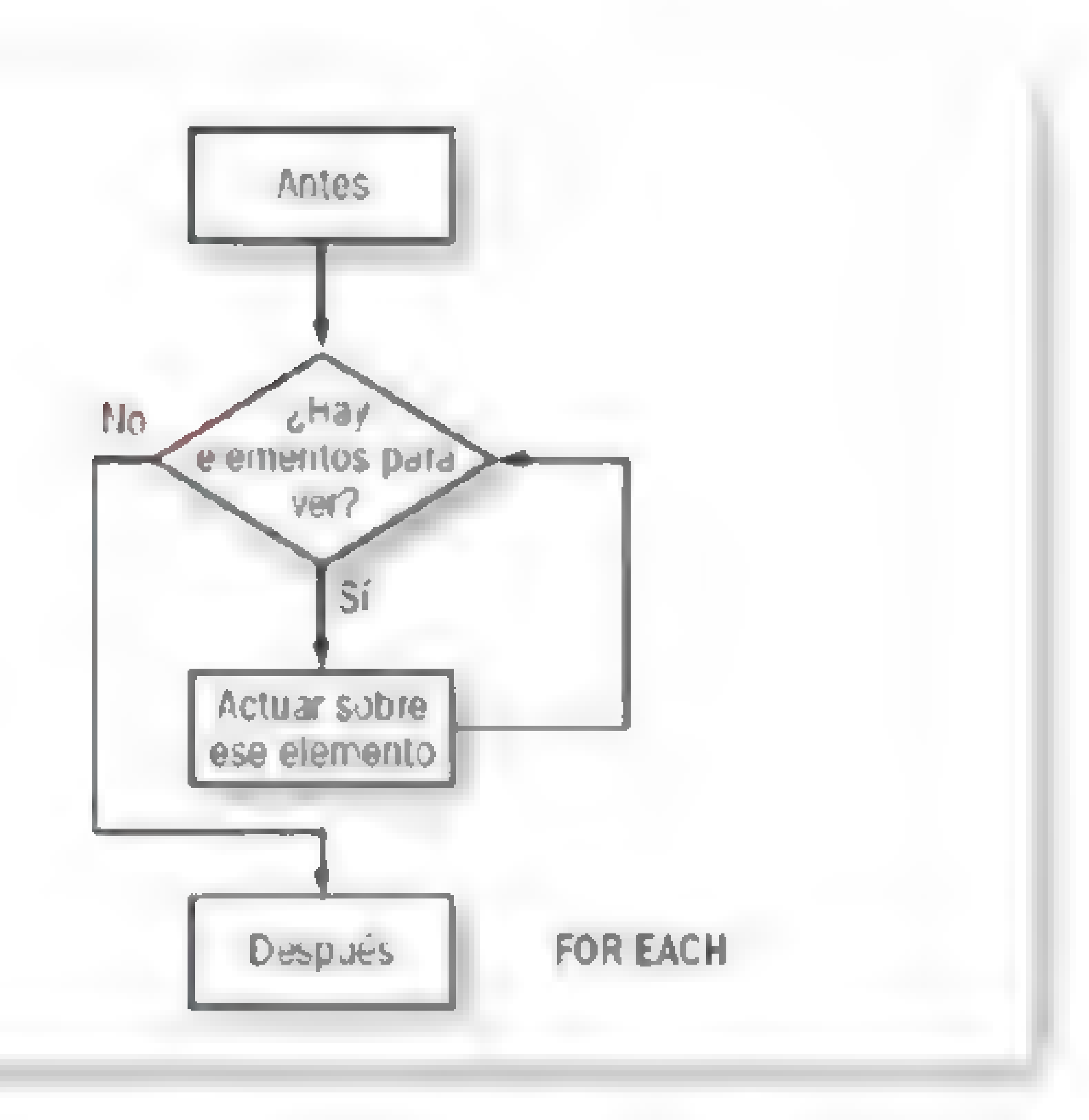


Figura 4. Diagrama de flujo correspondiente al ciclo for each.

Podemos decir que esta estructura es lo que se conoce como **syntax sugar** ('endulzar la sintaxis'), ya que no agrega funcionalidad sino solamente facilidad en el uso de algo que ya estaba; en este caso, del for común, como vemos en el siguiente fragmento:

```
int sum = 0;
for(Iterator i = digitos.iterator;
i.hasNext();
) {
int digito = ((Integer) i.next()).intValue();
        sum += digito;
}
assertEquals(sum, 45);
```

El programador tiene que escribir más código (y, por lo tanto, hay una mayor posibilidad de cometer errores) y es menos legible que la versión del for each. Esta es la traducción del for each al for que realiza el compilador Java y que, hasta la versión 1.5, los programadores tenían que escribir.

En la inicialización se crea una variable del tipo iterator y se la asigna con el iterador de la colección de los dígitos. La condición pregunta si existe un próximo elemento que visitar, si existe otro digito. El paso siguiente se encuentra vacío, dado que lo realizaremos en el cuerpo con el envío del mensaje next al iterador. En el cuerpo, conseguimos el siguiente elemento con next, se lo asignamos a la variable digito (que estamos creando en cada paso del for each) y realizamos la suma como en todos los ejemplos anteriores.

Forzamos el resultado de next a int con la instrucción (Integer) y el mensaje intValue. Lo primero se conoce como **casteo**, donde le indicamos a la JVM que sabemos el tipo real del objeto y queremos usarlo (next devuelve un object y no se puede asignar directamente a int). Lo segundo es realizado automáticamente por Java, en el caso del for each, y se conoce como **auto boxing** (envolver y desenvolver un dato primitivo en su clase asociada).

Declaración de variables, expresiones, sentencias y bloques

Veremos a continuación las distintas piezas básicas del lenguaje para la construcción del código.

Variables

En Java se conoce como **variable** a todo colaborador interno (**variables de instancia** y **de clase**), a los externos (**parámetros**) y a los nombres locales en los métodos (**variables locales**).
Existen algunas reglas para nombrar a las variables:

- El primer carácter debe ser una letra, el guión bajo (_) o el signo de dólar ($, aunque está reservado). Puede estar seguido de letras, números, el guión bajo o el signo dólar.
- No hay límite para la longitud del nombre.
- No debe ser una palabra clave.

Lo más conveniente es que los nombres de las variables reflejen el significado del dato que contienen (por ejemplo, conviene usar "velocidad" en vez de "v"). Si el nombre consiste en una serie de palabras, la primera letra de cada una de estas (salvo en la primera palabra) va en mayúscula (por ejemplo: diaDelMes). Se trata de una convención en Java, no una imposición. Por otro lado, las constantes se escriben totalmente en mayúsculas, separando las palabras por guiones bajos.

Una variable se define especificando primero su **tipo** y luego su **nombre**. Al mismo tiempo, se le pueden aplicar modificadores como "final", que hace que la referencia o el valor (se indica antes del tipo) solamente se pueda asignar una vez.

Expresiones

Una **expresión** es una combinación de **variables**, **operadores** y envíos de **mensajes que evalúan** un valor. El tipo del valor dependerá de los tipos de los elementos involucrados. Veamos algunos ejemplos:

```
speed = auto.getSpeed()
```

Aquí tenemos varias expresiones. Primero, auto, que devuelve el objeto referenciado (supongamos, de tipo **car**); luego el envío del mensaje getSpeed a auto, que devuelve un double, y, finalmente, la asignación, que

CASTEOS

Los casteos en Java no son bien vistos, ya que indican un error en el modelado. Hay casos particulares donde su uso es inevitable; uno de ellos, cuando se utilizan las colecciones de forma no genérica (código viejo), donde solo se utiliza Object.

devuelve el valor que se le asigna a speed y el tipo (que es del tipo de la variable). Veamos algunos ejemplos:

```
1 + 2 * 3
"hola" + ' ' + "mundo"
true != false
esPar(3) ? "par": "impar"
```

Sentencias

Las **sentencias** son la unidad completa de ejecución. En general, finalizan con un punto y coma (;) que las separa, salvo en el caso de los ciclos y las estructuras. Veamos algunos ejemplos:

```
speed = auto.getSpeed();
Fruta naranja = new Naranja();
unValor++;
auto.stop();
```

Bloques

Un **bloque** es una secuencia de sentencias encerradas entre llaves ({ y }) y puede ser usado en cualquier lugar donde va una sentencia.

Los bloques, en general, se utilizan como el cuerpo de las estructuras (como los ciclos), aunque también se los puede utilizar solos, ya que

VALORES POR DEFECTO

Toda variable en Java tiene un valor por defecto al momento de su creación. Los tipos primitivos **numérico** y **carácter** se inicializan en 0. Los datos de tipo **boolean** se inicializan en false. Cualquier referencia a objeto se inicializa en null.

definen un alcance léxico para los nombres. Esto quiere decir que se puede redefinir una variable en un bloque, así el código del bloque accede a esta y el código fuera del bloque accede a la primera definición.

Otras estructuras

Las **estructuras** son construcciones puramente sintácticas que se traducen en funcionalidad. Un ejemplo de estructura son los ciclos, vistos más arriba. Conozcamos otras estructuras que ofrece Java.

if/else/else if

Esta es una de las estructuras de control de flujo más básicas de los lenguajes de programación. Permite, sobre la base de una condición, modificar el flujo de ejecución, y se utiliza para situaciones del tipo de "Si llueve, no salgo, si no, salgo". Veamos cómo se escribe.

```
...
if(estaLloviendo()) {   //Si da false continúa en A
    noSalgo();
    ...                 //Al finalizar continúa en A
}
...                     //A
```

BLOQUES Y SENTENCIAS

Los ciclos y las estructuras como el `if`, que requieren un bloque de código para trabajar, pueden recibir una única sentencia en vez de un bloque. En estos casos, es posible obviar las llaves (aunque, por un tema de claridad, es mejor incluirlas siempre).

En el if se evalúa la condición booleana que se encuentra entre los paréntesis y, de acuerdo al resultado, se ejecuta el cuerpo de if (en caso de true). Si no, continúa con la ejecución ignorando el código de if.

Si queremos que se ejecute un código en el caso de false (manteniendo el caso de true), utilizamos la etiqueta else después del cuerpo de if, seguido de un bloque de código.

```
...
if(estaLloviendo()) {   //Caso true
    noSalgo();
    ...                 //Al finalizar continúa en A
} else {                //Caso false
    salgo();
    ...                 //Al finalizar continúa en A
}
...                     //A
```

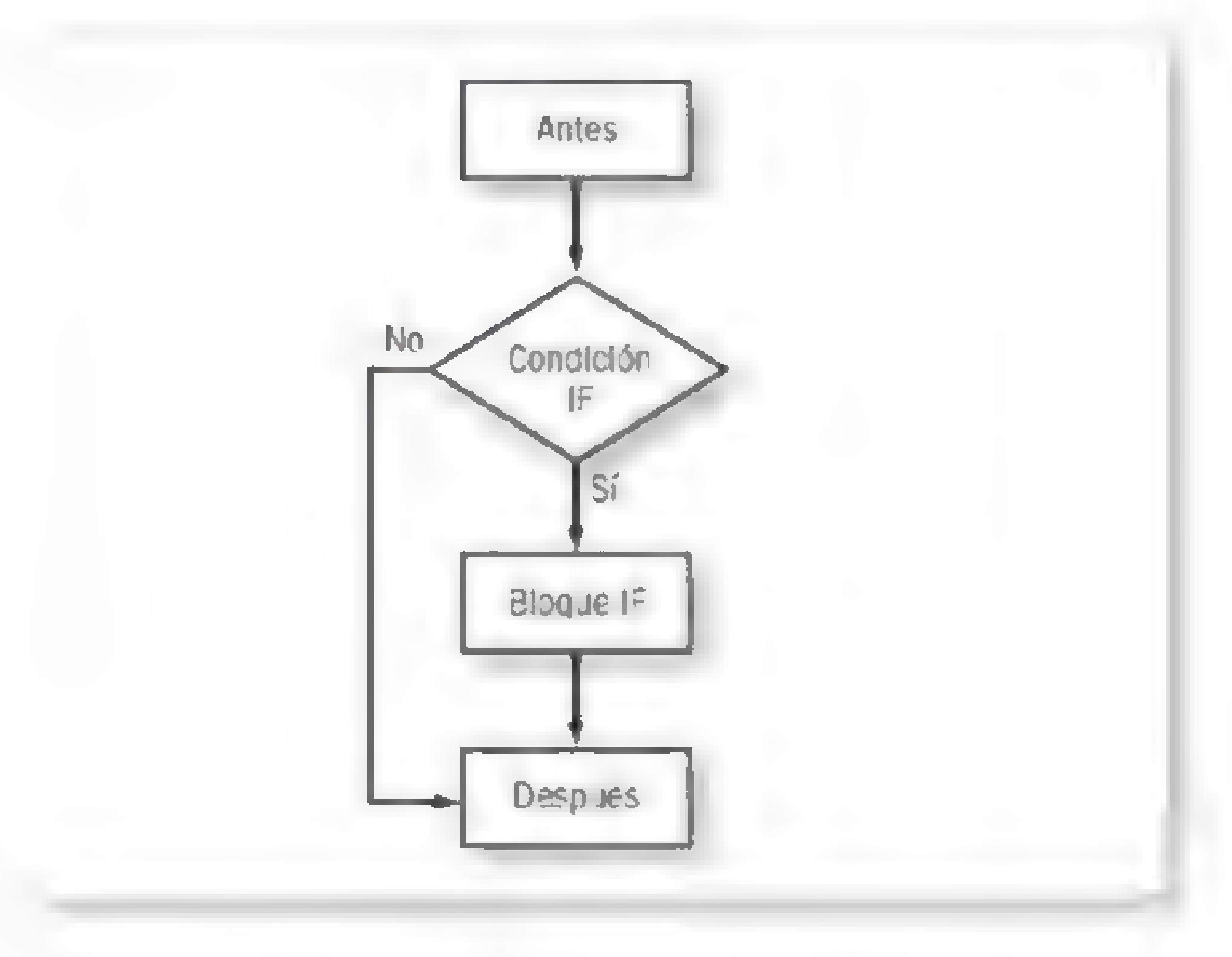


Figura 5. Diagrama de flujo correspondiente a if.

Finalmente, podemos encadenar varios if utilizando else if(<condicion>).

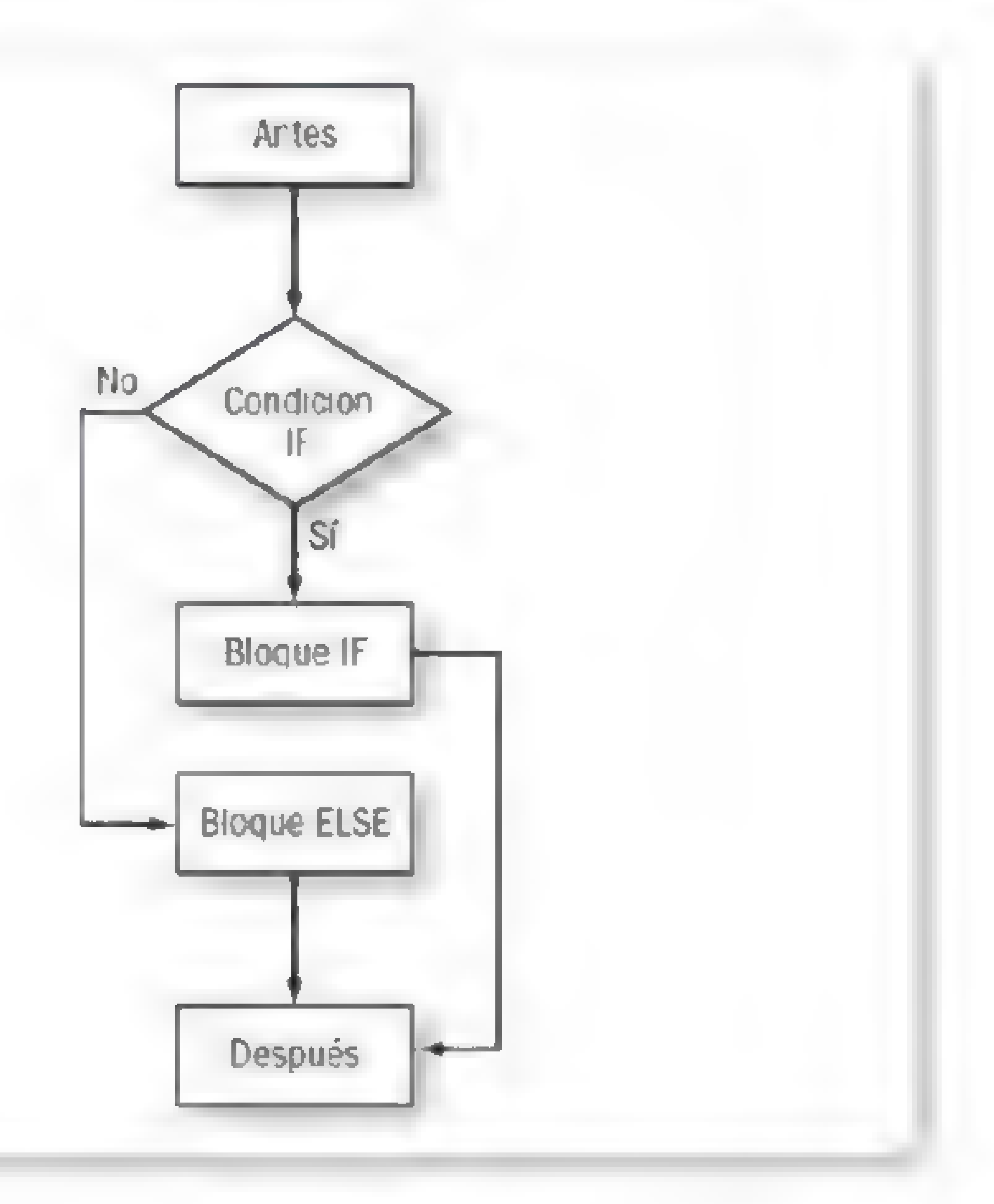


Figura 6. Diagrama de flujo correspondiente al if/else.

switch

Esta estructura de control de flujo permite comparar un tipo numérico entero y carácter primitivo contra varios candidatos, y ejecutar el código en consecuencia. La forma de escribirlo es:

Figura 7. Diagrama de flujo correspondiente al switch.

La forma de escribirlo es la que vemos a continuación.

```
switch(<valor>){
        case <literal>: <código> break;
        case <literal>: <código> break;
        ...
        default: <código> break;
}
```

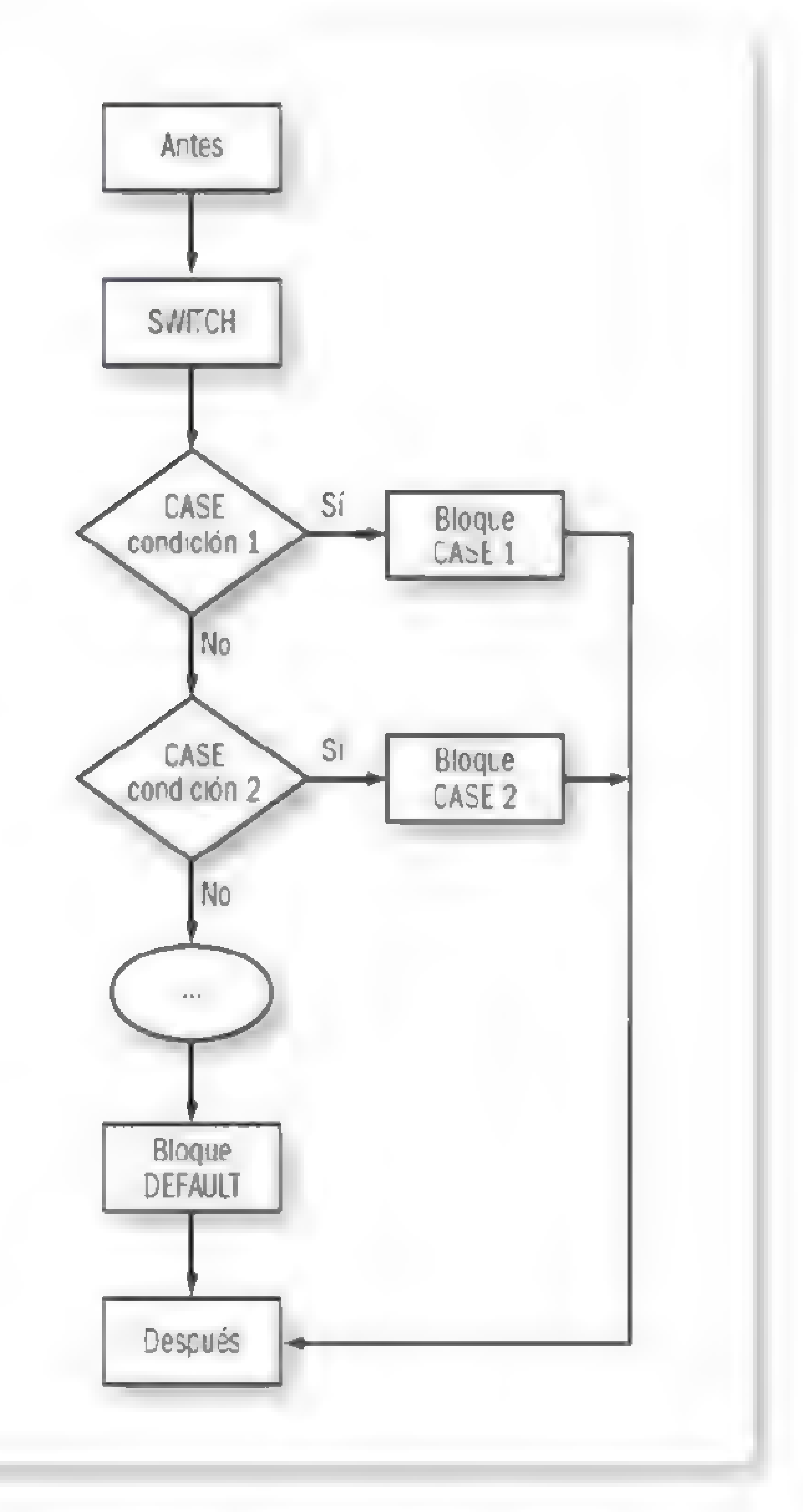


Figura 7. Diagrama de flujo correspondiente al `switch`.

A switch le pasamos un valor para comparar. Luego, en cada caso, lo comparamos contra un literal del mismo tipo y, de ser iguales, se ejecuta el código asociado. Se coloca la etiqueta break al final del código de cada caso para poder finalizar la ejecución del switch (generalmente es lo deseado),

ya que, si no, se seguirían ejecutando en cascada el resto de los códigos del switch (generalmente es un bug). El caso default se ejecuta cuando ninguno de los casos coincidió.

```
int opcion = 2;
switch(opcion) {
            case 1: ...     //Código para el caso 1
                            //Sin break, continúa ejecutando el caso 2
case 2: ...                 //Código para el caso 2
            break;          //Continúa en A
            default: ...    //Código para cuando no es ni 1 ni 2
            break;          //Continúa en A
}
...                         //A
```

try/catch/finally

Esta estructura está generalmente asociada al manejo de excepciones, aunque puede ser usada para otros fines. El código del bloque se ejecuta seguido de try, y luego puede suceder que el código se ejecute exitosamente, o bien que ocurra una excepción.

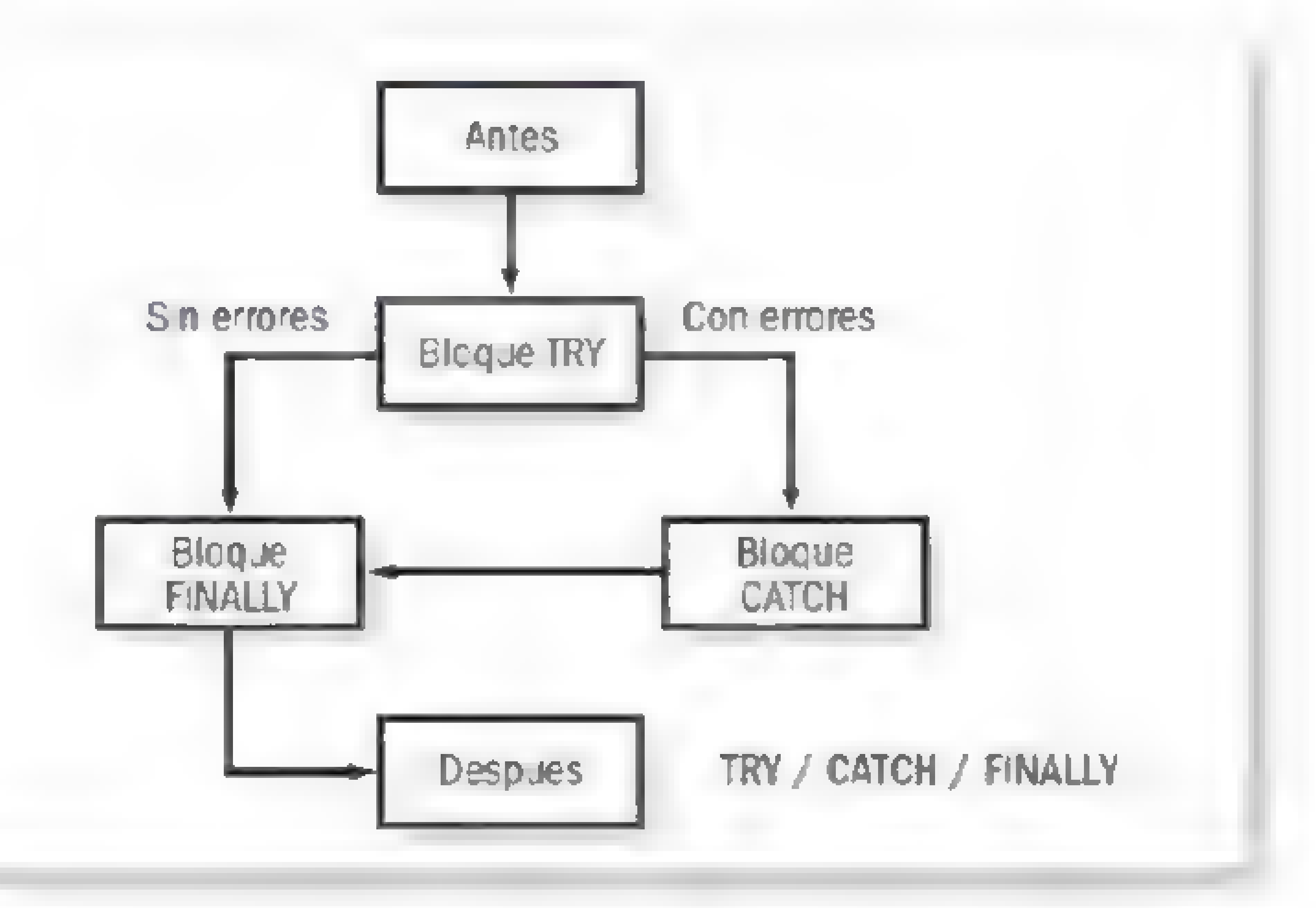


Figura 8. Diagrama de flujo correspondiente al try/catch/finally.

Si se especificó atrapar una excepción con la etiqueta catch, entonces se ejecutará el código asociado a esta. Si finalizó bien la ejecución o se terminó de ejecutar el código de catch, y se especificó código con la etiqueta finally, se evalúa este.

```
...
try {
            ...
            throw new Exception("Ocurrió un error");
//Continúa en B
            ...     //Si finaliza bien continúa en A
} catch(Exception e) {
            ...     //B, continúa en A
}
...                 //A

...
try {
            ...
            throw new Exception("Ocurrió un error");
//Continúa en B
            ...     //Si finaliza bien continúa en B
} finally {
            ...     //B, continúa en A si terminó bien
}
...                 //A

...
try {
            ...
            throw new Exception("Ocurrió un error");
//Continúa en B
            ...             //Si finaliza bien continúa en C
} catch(Exception e) {
            ...             //B, continúa en C
} finally {
            ...             //C, continúa en A si terminó bien
}
...                         //A
```

synchronize

Esta es una estructura de sincronización de **threads**. Se utiliza para asegurarse de que un solo thread está ejecutando el código del bloque asociado. Para ello, se usa un objeto (cualquiera) a modo de llave, y solo un único thread puede tener la llave en un momento determinado. El que la tiene es el que puede ejecutar, y el resto espera hasta que se libere la llave.

```
Object key = new Object();
...                         //Este código puede ser
ejecutado en paralelo por varios threads
synchonize(key) {
      ...                   //Este no
}
```

Tipos primitivos y literales

Si bien Java es un lenguaje orientado a objetos, posee valores que no son objetos, llamados **tipos primitivos**. Estos son representantes de los números y caracteres que, debido a problemas en la performance de las primeras versiones de Java, se decidió que no fueran objetos.

Son valores bien conocidos por la máquina virtual, y sus operaciones están cableadas en ella. Esto presenta un desafío para el programador, que debe lidiar con objetos y tipos primitivos. Al no ser objetos (es decir, al no ser referenciados) no se les puede asignar null, ni pueden ser utilizados en lugar de cualquier objeto. No son instancia de ninguna clase, no tienen métodos ni responden a mensajes. Solo se puede realizar operaciones sobre ellos como la suma y la resta, y otras definidas por el lenguaje directamente. Para tratar de salvar un poco estos problemas, Java introduce clases

que envuelven a los tipos primitivos en objetos que, además, incluyen algunos métodos para transformar del tipo primitivo a texto y viceversa. Los tipos primitivos son:

- boolean: representa los valores **verdadero** y **falso** de la **lógica de Boole**. Los valores de este tipo son true y false. La clase asociada con este tipo primitivo es Boolean.

```
boolean verdadero = true;
```

- byte: representa los números de ocho bits entre -128 y 127. La clase asociada es Byte.

```
byte eñe = (byte)164;   //ASCII de la letra ñ
```

En Java, los números enteros se pueden escribir tanto en **decimal** (164) como en **octal** (0244) y en **hexadecimal** (0xA4 o 0xa4).

- char: es el tipo que representa a un carácter Unicode (16 bits). La clase asociada es Character. Los literales de carácter en Java se escriben entre comillas simples, y aceptan un carácter o el código Unicode. También aceptan un número.

```
char eñeCaracter = 'ñ';
char eñeNumero = 241;  //Código Unicode decimal
char eñeUnicode = '\u00F1';
```

Para caracteres como el retorno de carro y la sangría, existen códigos especiales que sirven también para las cadenas de texto, denominados **caracteres escapados**.

```
char lineaNueva = ‘’;
char comillaSimple = ‘\’’;
char comillaDoble = ‘\”’;
char lineaNueva = ‘’;
char sangria = ‘\t’;
char barraInvertida = ‘\\’;
```

- double: es el tipo de los números de punto flotante de 64 bits de precisión definidos en el estándar **IEEE 754**, cuya clase asociada es Double. Los literales se pueden escribir con los decimales (usando el punto como separador) o en notación científica.

```
double unValor = 123.4;
double otroValor = 1.234e2; / Mismo valor pero en notación científica
double yOtroMas = 123.4d;   / Especifico que es double el literal
```

- float: similar al double pero de 32 bits de precisión. Su clase asociada es Float. Los literales se pueden escribir como los de double (sin la d), aunque puede surgir un problema al transformar un double literal a un float; lo conveniente es especificar que se trata de un literal de float usando la letra f.

```
float literal = 123.4f;
```

- int: representa los números de 32 bits enteros, entre -2147483648 y 2147483647. Su clase asociada es Integer. Los literales se pueden escribir de la misma forma que los literales de byte.
- long: representa los números de 64 bits, entre -9223372036854775808 y 9223372036854775807. Su clase asociada es Long.
Los literales se pueden escribir en decimal, octal o hexadecimal.

- short: representa los números de 16 bits, entre -32768 y 32767. Su clase asociada es Short. Los literales se pueden escribir de la misma forma que los literales de long.
- String si bien String no es un tipo primitivo, ya que es una clase, la incluimos en este apartado dado que el lenguaje da soporte para manejar las cadenas de caracteres de forma similar a un dato primitivo. En los literales aplican también las reglas de escape vistas para los char y se crean utilizando la comillas dobles (“”).

```
String texto = “hola mundo”;
```

- Array: son colecciones indexadas de elementos de tamaño fijo y establecido en el momento de creación. El acceso a los elementos de un array está dado por un índice que va desde 0 hasta la posición n-1, donde n es la longitud del array. Los arrays son instancias de la clase Array, y se pueden crear de la siguiente manera:

```
String [] unArray = new String[4]; // Un array con 4 posiciones vacías
String [] otroArray = new String [] {“a”, “b”};
```

El acceso a los elementos de los arrays se realiza con el operador [] y un índice entero:

```
assertEquals(otroArray[0], “a”);
otroArray[0] = “c”;
assertEquals(otroArray[0], “c”);
otroArray[10] = “error”; //Arroja una excepción de
tipo ArrayIndexOutOfBOundException
```

Operadores

Java permite operar sobre los tipos numéricos primitivos, usando los conocidos operadores matemáticos, más algunos otros. Estos operadores realizan funciones que están cableadas en la máquina virtual de Java y no son envíos de mensajes. Los operadores son:

Operadores aritméticos

- +, -, /, *: suma, resta, división, multiplicación.
  El operador + también sirve para concatenar varias strings.
- %: módulo.
- -: negación de un número.

Operadores lógicos

- &, &&, |, ||, ^, !: y, y (**short circuit**), o inclusivo, o inclusivo (**short circuit**), o exclusivo, negación.
- ==, !=: igualdad, desigualdad. Evalúan la igualdad de tipos primitivos (por ejemplo, 4 == 3 + 1 da como resultado true) o de referencias para los objetos, si una variable hace referencia al mismo objeto. Pero no verifican que dos objetos sean iguales (intercambiables), ya que esto se realiza enviando el mensaje equals a un objeto, mediante el envío del objeto a comparar.
- <, <=, >, >=: menor, menor o igual, mayor, mayor o igual. Sirven para comparar valores numéricos primitivos entre sí.

SHORT CIRCUIT LÓGICO

Esta técnica es muy utilizada cuando se quiere enviar mensajes a un objeto en una condición, pero no se sabe si la referencia está apuntando correctamente (o sea, no a null). La estructura es if(referencia != null && referencia.hayQueSeguir())

Los operadores de **short circuit** se utilizan cuando no se quiere evaluar la segunda parte de la condición, dado que el resultado de la primera parte es suficiente para saber el resultado general.

Operadores de bit

- &, |, ^: y, o inclusivo, o exclusivo.
  Operan sobre los bits que conforman los tipos primitivos.
- ~: complemento bit. Invierte los bits de un valor.
- <<, >>, >>>: operadores de corrimiento de bits. Mueven los bits hacia izquierda o derecha, tantos bits como se indiquen. En la práctica, es igual a multiplicar o dividir por 2 (mantienen el signo, excepto el operador >>>).

Otros operadores

- ++, --: incremento o disminución en 1 de una variable de tipo numérica entera (byte, short , int y long). Pueden ser tanto sufijos como prefijos (en el primer caso, primero se modifica el valor y luego se lo lee; en el segundo, se lee y luego se modifica).
- =: asignación. Se utiliza para asignar un valor a una variable. Devuelve el valor asignado.
- ?:: operador ternario. Se utiliza como forma corta de la estructura if/else.
- instanceof : operador que chequea si un objeto es instancia de una determinada clase (se consideran también las superclases).

Paquetes

PARA CREAR PAQUETES EN JAVA, UTILIZAMOS LA PALABRA CLAVE "PACKAGE"

Los **paquetes** son la manera en que Java organiza las clases en agrupaciones con sentido. Sirven para importar las clases usadas y, también, para definir qué clases que representan cosas distintas pero se llaman igual pueden coexistir (por ejemplo, la clase Punto para geometría y la clase Punto para gráficos 3D).

En Java, los paquetes siguen la misma estructura de directorios que contiene a los archivos fuentes. Para crear un paquete, solo debemos crear una clase que indique que pertenece a ese paquete, y respetar la estructura de directorios. Utilizaremos la palabra clave package seguida del nombre completo del paquete (paquetes padres y su nombre al final). Debe ser la primera instrucción del archivo fuente:

```
package padre1.padre2.padre3.nombre;
```

En el ejemplo, deberíamos tener la estructura de directorios padre1; dentro de este, un directorio padre2; dentro de padre2, otro llamado padre3; y, finalmente, dentro de este último, el directorio nombre con el archivo fuente de la clase.

Para importar todas las clases de un paquete, debemos declararlo utilizando import y el nombre completo del paquete, e indicando con un asterisco (*) que importamos todas la clases que están en el nivel indicado (no se importan las clases que están en subpaquetes).

```
import red.user.java.*;
```

También podemos indicar que importamos solamente una clase:

```
import red.user.java.ClaseParticular;
```

Las instrucciones de import deben ir después de la declaración de paquete y antes de la definición de la clase. Finalmente, podemos importar todos los métodos estáticos de una clase si agregamos el modificador static a la instrucción import de una clase.

```
import static red.user.java.ClaseConMetodosEstaticos;
```

RESUMEN

En este capítulo hemos conocido sintaxis y semántica del lenguaje Java: es decir, las reglas que permiten definir programas que sean correctos desde el punto de vista del lenguaje (compilador y máquina virtual). Vimos también que estas reglas, por sí solas, no hacen que un programa sea correcto en cuanto a la tarea que tiene que realizar, sino que somos los programadores los encargados de verificar que su lógica sea correcta.

Actividades

TEST DE AUTOEVALUACIÓN

1 ¿Para qué se usa final?

2 ¿Qué diferencias hay entre while y do while?

3 ¿Para qué sirve la estructura if/else/else if?

4 ¿Qué son las **expresiones**?

5 ¿Qué es una **sentencia**?

EJERCICIOS PRÁCTICOS

1 Cree un array, recórralo usando el ciclo for y opere sobre los elementos.

2 Cree un array, recórralo usando el ciclo for each y opere sobre los elementos.

3 Acceda a una posición inválida de un array. ¿Qué ocurre?

4 Sume los primeros diez números usando el ciclo for.

5 Sume los primeros diez números usando el ciclo while.

Clases

Las clases son los moldes para la creación de objetos, ya que definen la forma y el comportamiento de las instancias creadas. A lo largo de este capítulo, veremos cómo se crean y cómo se definen sus atributos y métodos. Será de gran utilidad para los lectores haber repasado antes la teoría sobre programación orientada a objetos.

Definición

Hemos visto cómo aparece una clase en nuestros Unit Test ; ahora, entenderemos en profundidad cada elemento de su definición. Todo archivo fuente Java requiere la existencia de una **clase pública** con el mismo nombre. A continuación, encontramos el caso más simple de una clase:

```
public class Auto {
        ...
}
```

Indicamos que es pública con el modificador public, luego aparece class y, finalmente, el nombre. Por defecto, toda clase hereda de Object, pero si queremos heredar de alguna otra, podemos especificarlo a continuación del nombre utilizando la palabra extends, seguida del nombre de la clase padre.

```
public class Auto extends Vehiculo {
        ...
}
```

Solamente se puede heredar de una clase: recordemos que Java implementa **herencia simple**. Tratando de alivianar esta restricción, Java utiliza las interfaces para definir protocolos homogéneos a través de jerarquías heterogéneas. Para decir que una clase implementa una interfaz (o varias), escribimos el nombre de esta (en el caso de ser varias, separados por comas) después de la palabra implements .

```
public class Auto extends Vehiculo implements VehiculoTerrestre {
        ...
}
```

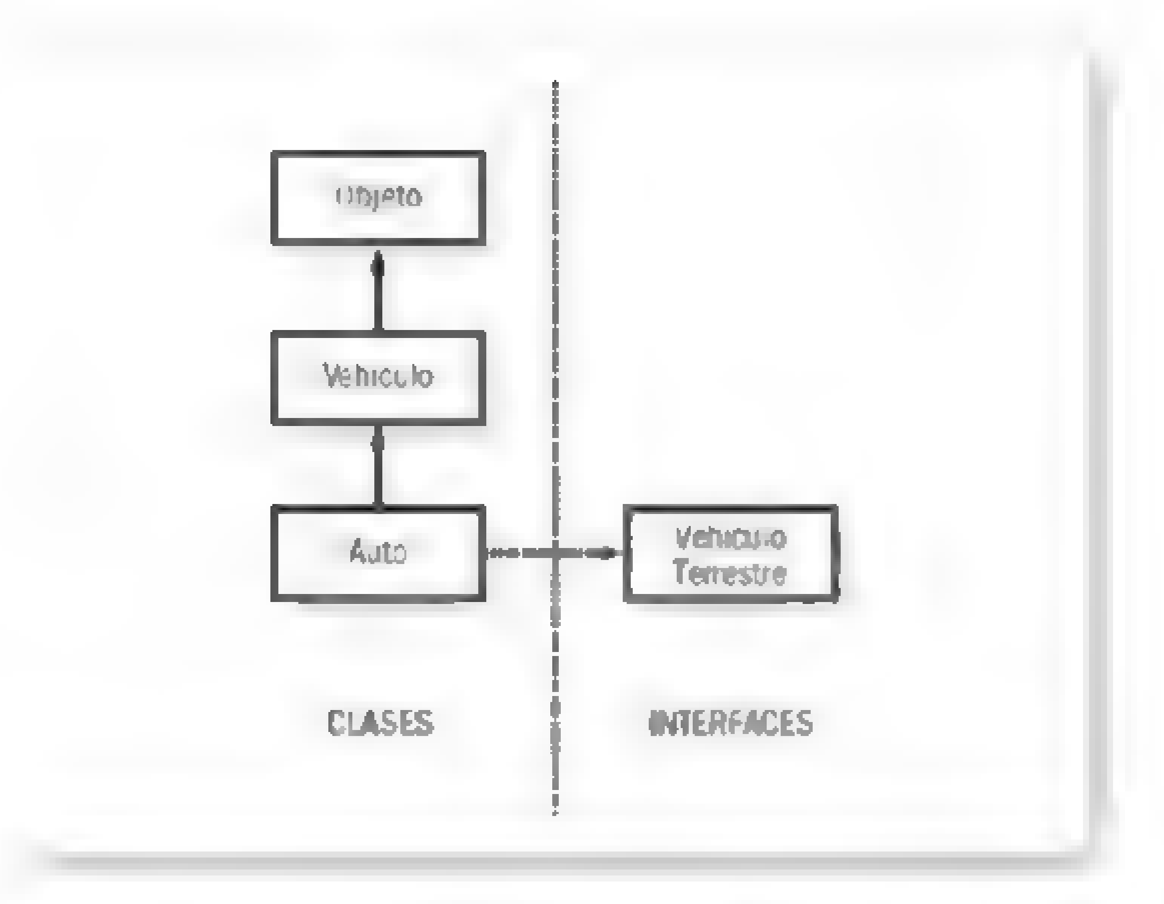


Figura 1. Relación de herencia entre las clases y de implementación con la interfaz.

La clase principal del archivo debe ser pública o, de lo contrario, no debe indicarse nada, ya que no puede ser privada; a este tipo de clases se las llama de **paquete**. Las clases con la visibilidad de paquete solamente pueden ser utilizadas por otras clases que estén dentro del mismo paquete. Las clases públicas pueden ser utilizadas por cualquier otra clase y también pueden ser marcadas como finales, utilizando el modificador `final` antes del `class` en la definición (este modificador hace que no pueda existir una subclase de ella). Finalmente, las clases pueden ser marcadas como abstractas (moldes para otras clases) usando el modificador `abstract`. Las clases abstractas no pueden tener instancias y sirven para agrupar el conocimiento y comportamiento en común de las subclases de este tipo.

En Java, las clases son instancia de la clase `Class`. En ella están definidos los mensajes que entiende cada clase, como el tipo de mensajes que entienden sus instancias o qué constructores tienen, entre muchas otras cosas.

Para acceder a la instancia de `Class` correspondiente a una clase podemos utilizar el nombre seguido de `.class`, o podemos pedírsela a una instancia mediante el mensaje `getClass()`. Ambas formas devuelven un objeto de tipo `Class`, que es el dato que nos interesa.

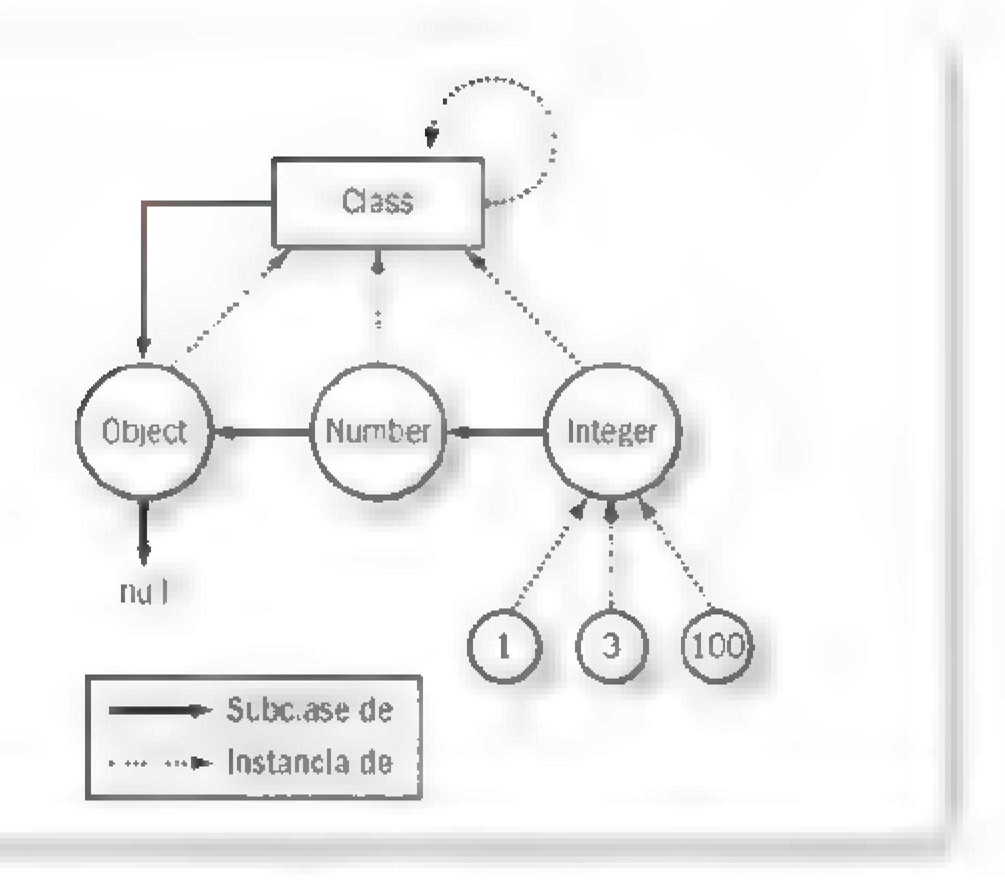
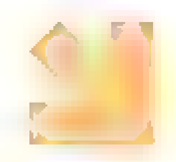


Figura 2. Vemos las relaciones entre las clases, la clase Class y las instancias.



Atributos

Los **colaboradores internos**, también conocidos como **variables de instancia** o **atributos**, se definen en el cuerpo de la clase. Su definición es similar a la de cualquier variable, con la particularidad de que se le pueden agregar los modificadores de visibilidad. Como buena práctica, siempre definiremos los atributos como privados, dado que son detalles de implementación que deberían estar siempre ocultos a los demás objetos (incluso a aquellos

PROPIEDADES Y JAVABEANS

Cuando un objeto tiene un getter y un setter públicos para un determinado atributo, se dice que tiene una **propiedad**. Llamaremos a la propiedad por el nombre del atributo. Esta es una convención nacida de los **JavaBeans**, objetos destinados a ser manipulados gráficamente, que nunca cumplieron su objetivo.

pertenecientes a alguna subclase). Al mismo tiempo, cuando queramos exponer un atributo como público, deberemos hacerlo a través de métodos para leer y para escribir (llamaremos getter al de lectura y setter al de escritura).

Existe la convención de nombrar a estos métodos utilizando los prefijos get, set o is, seguidos del nombre del atributo:

```
public class Auto {
        private Marca marca;
        ...
        public Marca getMarca() {
                return marca;
        }
        public void setMarca(Marca marca) {
        this.marca = marca;
}
        ...
}
```

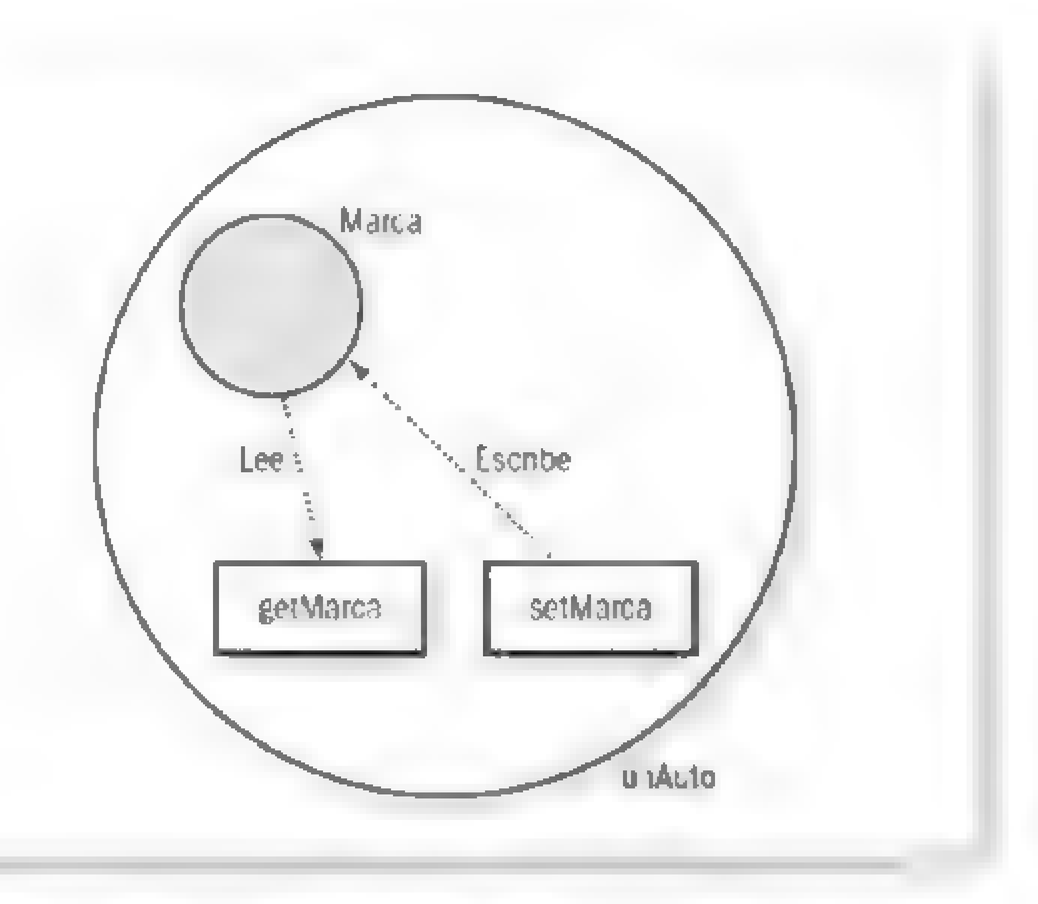


Figura 3. Vemos cómo se relacionan **getters** y **setters** con su atributo.

Las variables tipo boolean utilizan el prefijo is en vez del get.

```
public boolean isEmpty() {...}
public void setEmpty(boolean isEmpty) {...}
```

Esta es sólo una convención de Java, pero no quiere decir que sea obligatoria. Reflexionemos acerca del significado de un método que setea si un objeto esta vacío o no. ¿No tendría que ser, por ejemplo, vaciar()? ¿Y lo contrario, llenar()? Siempre pensemos el significado que queremos dar a los métodos en su contexto de uso. Lo importante es que utilicemos métodos para acceder a los atributos de un objeto, ya sea desde el exterior o del interior; esto nos dará un grado de flexibilidad muy importante.

Supongamos que tenemos un objeto Empleado, cuyo sueldo está representado por el atributo público sueldo. Cada vez que queramos obtener un dato del tipo sueldo, estaremos accediendo al atributo. Ahora supongamos que tenemos un cambio, y el sueldo, en vez de ser un valor fijo, depende de ciertos porcentajes variables. El sueldo tiene que ser calculado cada vez que se lo quiera leer. Tendremos que cambiar absolutamente todos los lugares donde se estaba leyendo el atributo sueldo por el envío de mensaje sueldo().

No solo por la adaptabilidad al cambio, sino también por la flexibilidad que da enviar un mensaje (ya que se decide en runtime por method lookup), es que debemos siempre enviar mensajes en vez de acceder

GETTERS Y SETTERS

En general es una buena idea seguir las convenciones de una comunidad: en este caso, utilizar **getters** y **setters** para acceder a atributos privados. Pero es mejor lograr una claridad de intención y de lectura que una convención. Pensemos los nombres de los métodos con cuidado, ya que reflejan nuestra intención a la hora de programarlos.

directamente a un atributo, incluso dentro del código de la misma clase. Las variables pueden ser inicializadas junto con su definición.

```
public class Auto {
        private Marca marca;   / Inicializado al valor por defecto (null)
        private String modelo = “”;   // Inicializado
        ...
}
```

En los métodos de la clase, estos atributos son accedidos directamente con su nombre o utilizando la pseudovariable this, que es una referencia al objeto actual.

Los atributos estáticos, aquellos que pertenecen a la clase, se definen agregando el modificador static. Pueden ser accedidos por medio de su nombre o utilizando el nombre de la clase junto al nombre del atributo (y, obviamente, dependiendo de la visibilidad especificada).

Cuando heredamos de una clase, heredamos los atributos definidos por ella, aunque no podamos acceder a ellos si están declarados como privados. Si estamos creando una jerarquía, es una buena práctica no definir atributos hasta que no lleguemos a las clases concretas (opuesto a lo abstracto: si consideramos que las clases abstractas están en lo alto de la jerarquía, las clases concretas estarían abajo). Si lo hiciéramos, estaríamos forzando una implementación particular y concreta en vez de una necesidad abstracta. En lugar de definir un atributo, deberíamos declarar métodos abstractos (protected o public, dependiendo de la finalidad). Así, las clases concretas pueden decidir ellas mismas qué implementación quieren y necesitan.

Métodos

Los **métodos** son la implementación asociada a un mensaje que entiende el objeto. Su definición consta de una firma que sirve para identificar al método, seguida del código. La firma o signatura de un método

se forma con los modificadores de visibilidad, seguidos de otros posibles modificadores (final, native, synchronized, static, etcétera); luego, el tipo de respuesta (o void, si no responde nada), el nombre (que sigue las reglas de los nombres de variables) y, finalmente, los argumentos que espera. Los argumentos se definen como variables (tipo seguido del nombre) con la posibilidad de aplicar el modificador final.

```
class Auto {
...
public Marca getMarca() {
        return this.marca;
}

public void arrancarCon(final Llave laLlave)
throws LlaveIncorrectaException
{...}
...
}
```

Si un método lanza excepciones, ya sea porque su propio código las lanza o no las atrapa, debe especificarlas como parte de la firma del mensaje e indicarlas luego de los argumentos, utilizando la palabra throws seguida de los nombres de las excepciones separados por coma.

Java permite que los mensajes tengan argumentos variables: por ejemplo, si queremos pasar un cierto número de parámetros en un envío y cierto número en otro, y queremos que, además, sea invocado el mismo método:

```
void cargarEquipaje(Equipaje ... equipajes) {
...
}
```

El envio del mensaje se realiza de la siguiente forma:

```
...
auto.cargarEquipaje(unEquipaje, otroEquipaje, yOtroEquipajeMas);
...
```

Se coloca la cantidad deseada de parámetros separados por comas. Esto es, en efecto, una facilidad del compilador, que transforma el código de la manera que sigue:

```
void cargarEquipaje(Equipaje [] equipajes) {
...
}
...
auto.cargarEquipaje(new Equipaje[] {unEquipaje,
otroEquipaje, yOtroEquipajeMas});
...
```

Como vemos, transforma todo del mismo modo en que sucedería si el mensaje recibiera un array y, en el código del método, accediéramos a los argumentos utilizando el parámetro como un array.

```
void cargarEquipaje(Equipaje ... equipajes) {
        for(Equipaje equipaje : equipajes) {
        this.cargar(equipaje);
}
}
```

Java permite que se puedan definir varios métodos con el mismo nombre dentro de una misma clase. Esto se conoce como **sobrecarga** (*overloading*). La restricción se halla en que los métodos difieran en la cantidad o tipo de argumentos que reciben.

```
void acelerarA(int kmPorH) {...}    // Válido
void acelerarA(double kmPorH) {...} // Válido
void acelerarA(int km, int h) {...}  // Válido
bool acelerarA(int kmPorH) {...}    // Inválido
```

Podemos ver un código de ejemplo en el archivo Automovil.java, disponible en el sitio **Premium.redusers.com**.

El compilador se encarga de decidir cuál mensaje se envía sobre la base de la información de tipos y cantidad de parámetros que se pasan.

Herencia y métodos

Cuando heredamos de una clase, nuestros objetos entienden todos los mensajes declarados en ella y en toda la ascendencia de clases. En la herencia, reutilizamos conocimiento y comportamiento al construir jerarquías. A medida que vamos extendiendo la jerarquía, ampliamos el conocimiento (agregando un nuevo comportamiento) o especializamos el comportamiento heredado. Agregar un nuevo comportamiento es, simplemente, definir nuevos mensajes al agregar nuevos métodos.

En cambio, cuando queremos especializar cierto comportamiento, lo que buscamos es que ciertos objetos se comporten de manera distinta a los objetos de clases padres al recibir el mismo mensaje (ya sea

USO DEL FINAL

Es una buena practica modificar tanto los argumentos de los metodos como las variables definidas en ellos con final. Así se evita la generación de errores al reasignarles valores a tales variables (o argumentos) en el método. Estos errores son generalmente difíciles de detectar

haciendo más acciones u otras distintas). Para lograrlo, redefinimos el mismo método (con la misma firma) en la clase que nos interesa.

```
public class Lista {
        public void agregar(Object objeto) {...}
}
public class ListaOrdenada extends Lista {
        @Override
public void agregar(Object objeto) {
        super.agregar(objecto);
        this.ordenar();
}
}
```

La anotación @Override es opcional, pero es una buena práctica agregarla, así, al compilar, el IDE puede avisarnos si no estamos redefiniendo un método, al habernos equivocado en la firma.

Como buena práctica, deberíamos utilizar siempre super para invocar el comportamiento que estamos especializando, generalmente como primero o último paso de un método. Si lo llamamos en medio de la ejecución, haremos algunos preparativos antes de llamar y luego procesaremos el resultado antes de devolverlo.

USO CORRECTO DEL SUPER

Cuando usamos super no tenemos restricciones sobre que mensajes enviamos. Una buena practica es restringir los envios con super al mismo mensaje que se está ejecutando. Así, el código es mas fácil de seguir (tanto para nosotros como para otros programadores). De este modo, cometeremos menos errores dificiles de detectar y corregir

Constructores

Los **constructores** son métodos especiales para inicializar las instancias de una clase. Se llaman igual que la clase, no tienen tipo de retorno (ni siquiera void) y solamente se les pueden aplicar los modificadores de visibilidad.

```
public class Celular extends Telefono {
        ...
public Celular() {...}
public Celular(NumeroTelefonico numero)
        throws NumeroTelefonicoInvalido {...}
        public Celular(
            Prefijo prefijo,
            NumeroTelefonicoLocal numeroLocal
) throws NumeroTelefonicoInvalido {...}
...
}
```

Los constructores también se pueden sobrecargar, como cualquier método, variando la cantidad de parámetros que reciben. Por otro lado, son invocados (no se trata de un envío de mensaje) cuando se crea una instancia. Normalmente, crearemos instancias mediante el operador new:

```
Utensilio tenedor = new Tenedor();
```

El operador new se encarga de crear una instancia de la clase, reservando espacio en la memoria para ella y para todos sus atributos. Luego, con la instancia creada, se invoca al constructor para que la inicialice. Java crea automáticamente un constructor sin parámetros y público, en caso de que el programador no haya declarado algún constructor en la clase.

Una clase padre fuerza los constructores en sus clases hijas. Por este motivo, una clase hija tiene que definir un constructor con la misma firma que la clase padre. Además, cuando una clase define un constructor, debe invocar algún constructor de la clase padre como primer paso en el código, a fin de asegurarse de que lo definido por la clase padre está bien inicializado antes de inicializar los atributos declarados por la clase hija. Esto se logra usando la palabra super (y los argumentos necesarios) al llamar al constructor padre.

```
public class Telefono {
        public Telefono(NumeroTelefonico numero) {...}
        ...
}
...
public class Celular extends Telefono {
        public Celular(NumeroTelefonico numero) {
                super(numero);
                ...
        }
        ...
}
```

Las clases hijas pueden definir otros constructores, además de los establecidos por la clase padre.

this y super

En los métodos de instancia (así como en los constructores) tenemos a nuestra disposición dos pseudovariables (ya que no están definidas en ningún lugar): this y super. this es una referencia al receptor del mensaje cuyo método está siendo ejecutado. Con this, tendremos acceso a todos los elementos del objeto, atributos, métodos y constructores, y su uso es opcional, excepto cuando se debe desambiguar algún nombre.

```
public class Celular extend Telefono {
        private Prefijo prefijo;
        ...
        public Celular(NumeroTelefonico numero) {
                this(numero.prefijo(), numero.local());
// Utilizo this para llamar al otro constructor
}
public Celular(
     Prefijo prefijo,
          NumeroTelefonicoLocal numeroLocal
) {
        super();    Llamo al constructor de la clase padre
        this.prefijo = prefijo; // Utilizo this para
indicar que accedo al prefijo de la instancia y no al argumento
        ...
}
...
public Prefijo getPrefijo() {
        return this.prefijo; //Acá el this es opcional
}
}
```

super, al igual que this, es una referencia al objeto receptor del mensaje cuyo método está siendo ejecutado. La diferencia con el anterior radica en que no se puede utilizar salvo para enviar mensajes, y en que los métodos son buscados a partir de la clase padre de la clase a la que pertenece el método que se está ejecutando. El código que aparece en la siguiente página ejemplifica el uso y funcionamiento de super. Encontraremos este código disponible para descarga en el sitio **Premium.redusers.com**.

SUPER

Muchos creen que super significa **superclase** del objeto receptor. Donde las jerarquías son complicadas y se sobrescriben muchos métodos, puede llevar a errores difíciles de encontrar.

```
public class A {
        public String bar() {
                return "A";
        }
        public String zip() {
                return "A";
        }
public A getThis() {
                return this;
        }
}
...
public class B extends A {
        public String foo() {
                return super.bar();
        }
        @Override
        public String bar() {
                return "B y " + super.bar();
        }
}
...
public class C extends B {
        @Override
        public String bar() {
                return "C y " + super.bar();
        }
        @Override
        public String zip() {
                return "C y " + super.zip();
        }
}
...
public class SuperUnitTest {
        @Test public void testSuper() {
                C c = new C();
                assertSame(c, c.getThis());
                assertEquals("A", c.foo());
                assertEquals("C y B y A", c.bar());
                assertEquals("C y A", c.zip());
        }
}
```

Métodos estáticos versus no estáticos

Puede parecer que los métodos y atributos que definimos como estáticos pertenecen a la clase, ya que no pertenecen a las instancias y los accedemos mediante el nombre de la clase. Pero, en efecto, si recordamos que todas las clases son instancias de la clase Class, entenderemos que todas ellas responden a los mismos mensajes y poseen los mismos atributos. Entonces, ¿a dónde pertenecen los elementos estáticos? Están asociados a la clase, pero no pertenecen a esta; el compilador, junto con la JVM, se encarga de manejar el acceso a ellos. Se dice que son elementos **globales**, ya que cualquier código puede acceder a ellos (si tienen visibilidad), son resueltos en tiempo de compilación y no resultan del envío de un mensaje. Es posible acceder a los métodos estáticos de una clase padre usando el nombre de una clase hija. En ellos, no es posible usar this o super, ya que no hay instancia ni una jerarquía asociada.

En cambio, los métodos de instancia son resueltos en tiempo de ejecución y resultan del envío de un mensaje. Esto da una flexibilidad que no se tiene con los métodos estáticos, ya que el comportamiento está dado por cómo relacionamos los objetos durante la ejecución, en vez de forzarlo en tiempo de compilación a un código específico. Los envíos de mensajes son puntos de acceso a distintos comportamientos, ya que el objeto receptor puede cambiar, mientras que en los métodos estáticos está fijo para siempre, a menos que cambiemos el código fuente y lo recompilemos.

ES RECOMENDABLE TRATAR CON OBJETOS Y DEJAR DE LADO LO ESTÁTICO

La recomendación, siempre, es tratar con objetos y dejar de lado lo estático. Al hacerlo, estamos siendo más tolerantes al cambio, y recordemos que los cambios siempre existen. Podemos ver algún código de ejemplo en

los archivos NumeroComplejo.java y Persona.java, que se encuentran disponibles en el sitio **Premium.redusers.com**.

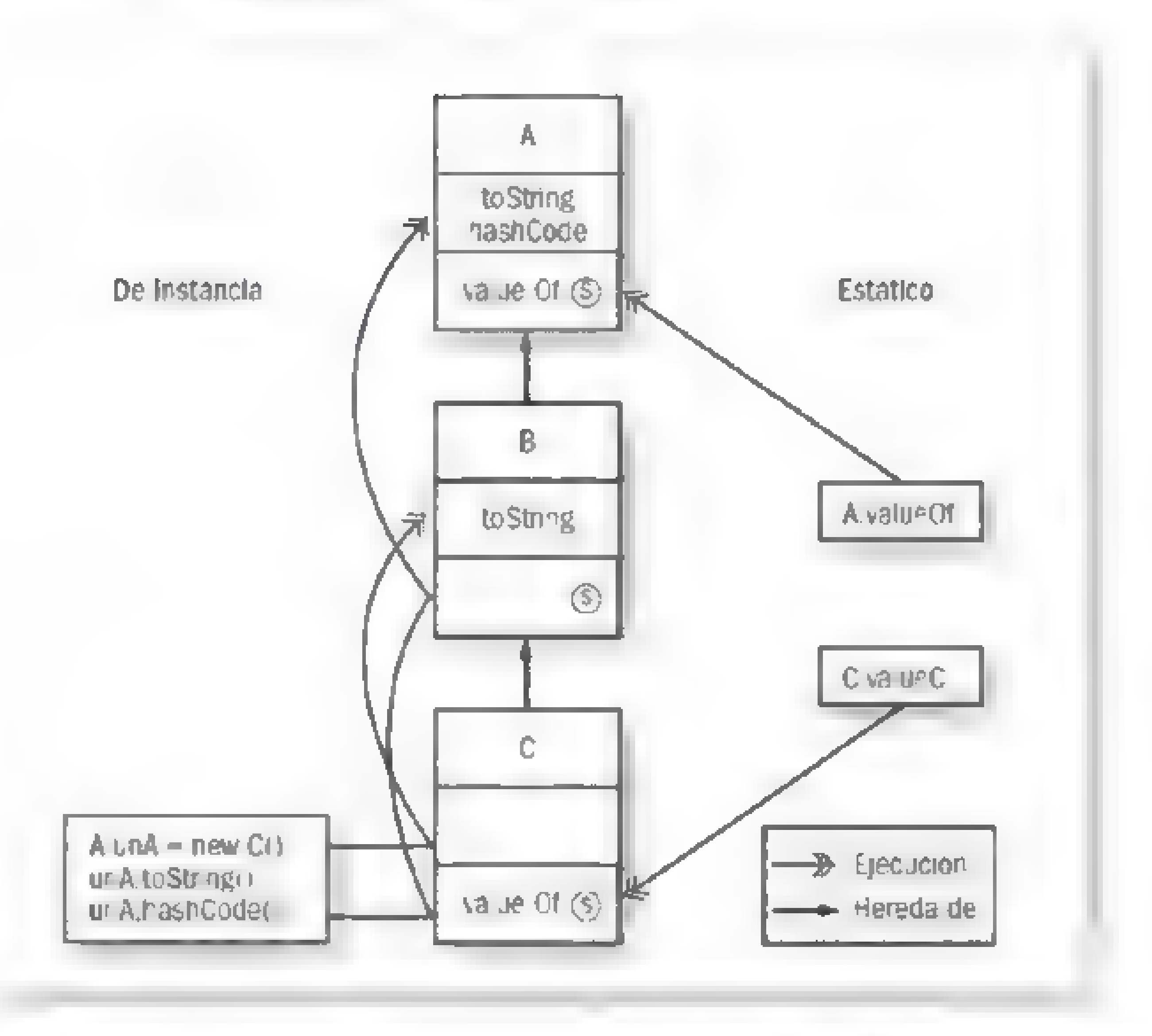


Figura 4. Diferencia entre ejecutar un método estático y un envío de mensaje.

Uso avanzado de clases

En los siguientes párrafos, veremos conceptos avanzados sobre las clases y su uso. Abarcaremos las nociones de **clase abstracta**, **clase anidada** y, finalmente, **clase anónima**. La clase abstracta representa un molde para otras clases, al agrupar conocimiento para formar una jerarquía. Las clases anidadas son clases que se definen dentro de otra.

Las anónimas son clases anidadas que no poseen nombre y que son definidas dentro de los métodos para un uso puntual. Tendremos a disposición un proyecto: el juego de la vida (disponible en **Premium.redusers.com**)

Clases abstractas

En apartados anteriores aprendimos la utilización básica de las clases en Java. Vimos cómo se definen y cómo se declaran los atributos, métodos y constructores. Conocimos y entendimos la diferencia que existe entre utilizar atributos y métodos de instancia frente a los estáticos, y la razón por la cual es importante no utilizar estos últimos.

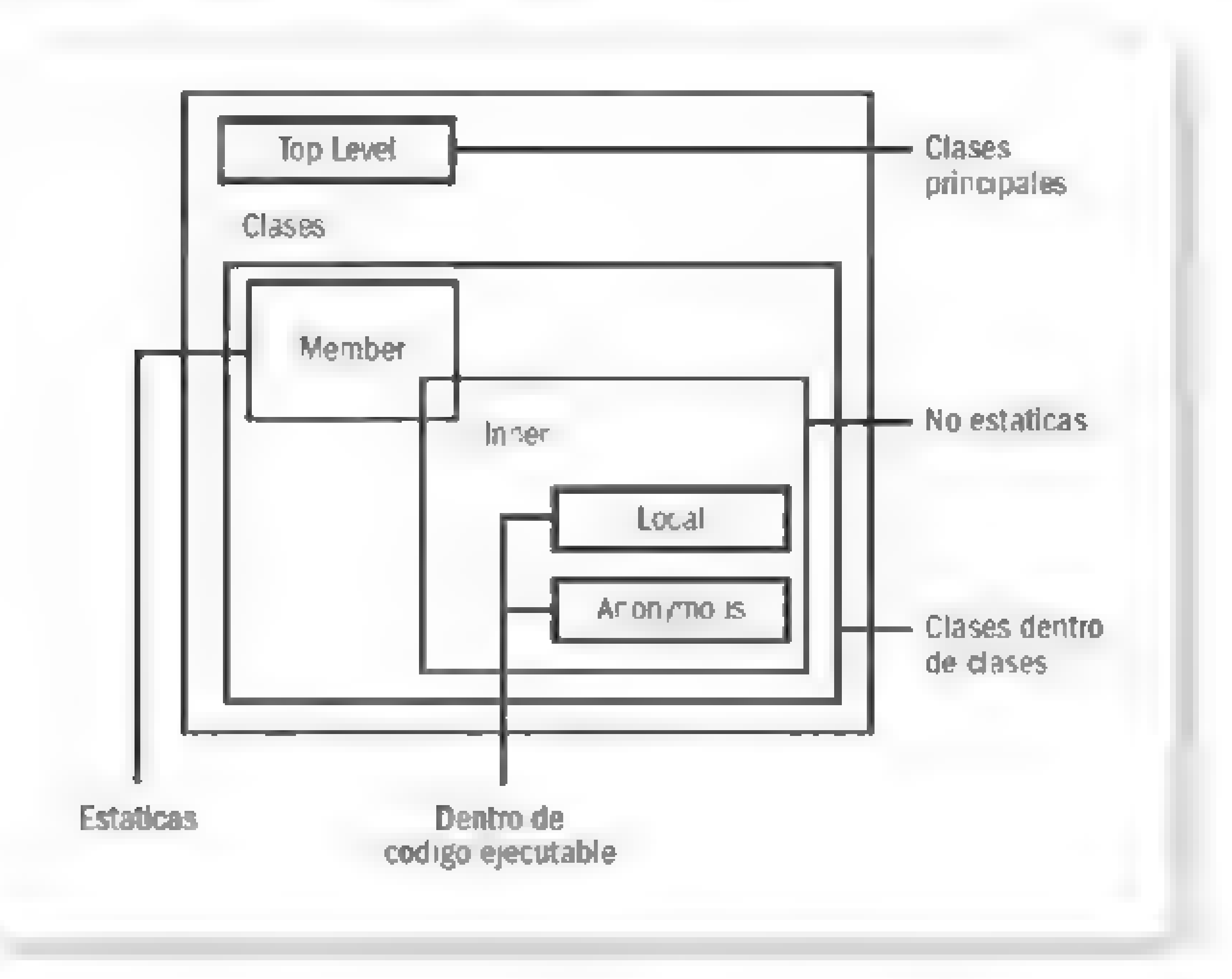


Figura 5. Esquema con los distintos tipos de clases y cómo se relacionan entre sí.

Tomemos como punto de partida la taxonomía de los animales. Encontramos, por ejemplo, los que son mamíferos y los que no lo son. Dentro de la categoría de los mamíferos están, a su vez, los caninos y, dentro de los caninos, los perros. En esta clasificación, "mamíferos" y "caninos" son categorías que agrupan animales que poseen ciertas características comunes entre sí.

Las **clases abstractas** son el equivalente a estas categorías, y definen el comportamiento esperado para un conjunto de objetos. Estos objetos pertenecen a una subclasificación de la clase abstracta, ya que una clase abstracta no puede tener instancias. Pensémoslo de este modo: no existe un mamífero

que no pertenezca a ninguna subclasificación; por ejemplo, los perros son caninos, que a su vez son mamíferos. No existe un animal al que llamemos simplemente mamífero. Así, un perro, por ejemplo, posee todas las características que lo definen como mamífero.

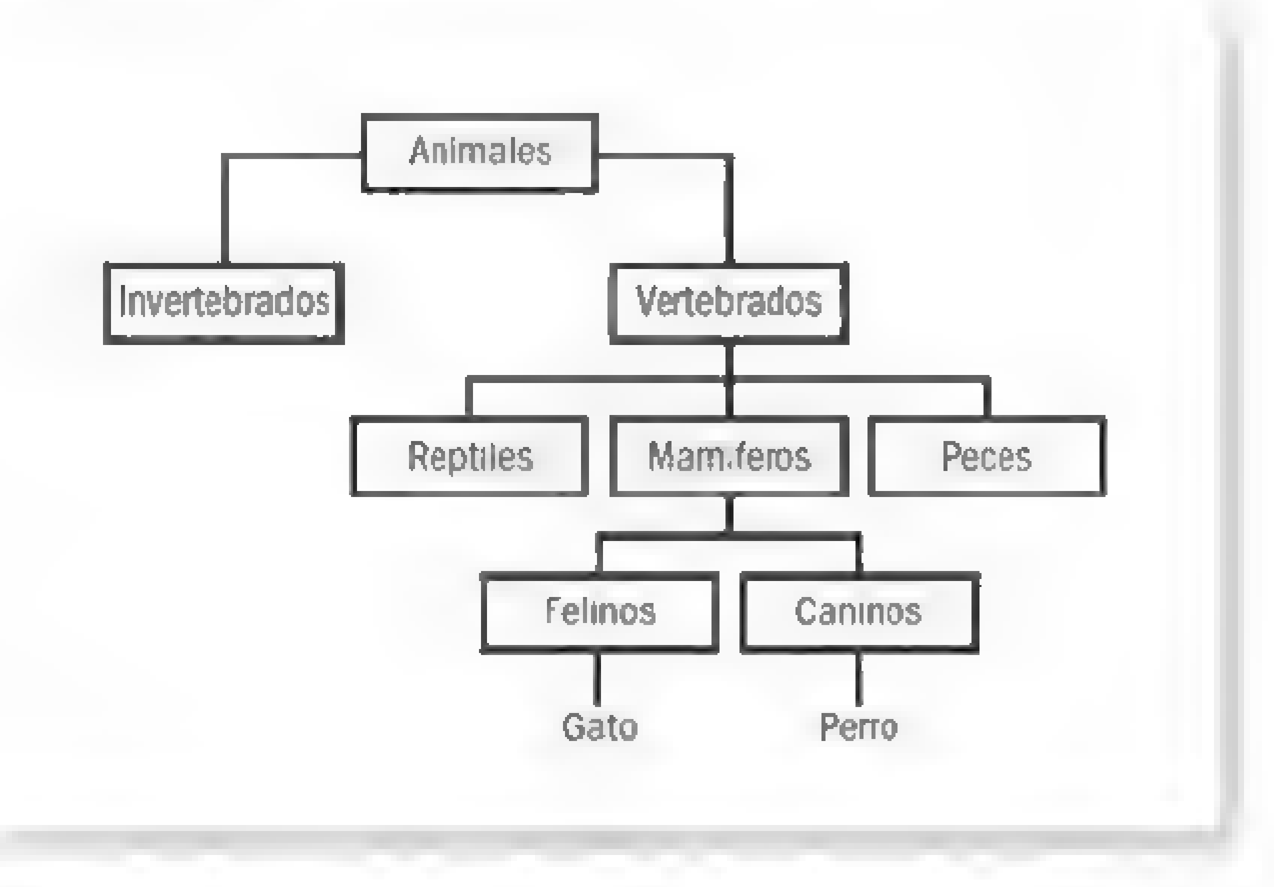


Figura 6. Posible categorización (incompleta) de los animales.

En Java, las clases abstractas se declaran agregando el modificador abstract a la definición:

```
public abstract class Mamifero {
        ...
}
```

Pueden especificar métodos sin definirles código. Estos son los llamados **métodos abstractos**, que declaran un comportamiento esperado para los objetos que sean de ese tipo y dejan la implementación específica a las subclases. Los métodos abstractos se definen con el modificador abstract.

```
public abstract class Mamifero {

    abstract void amamantarA(Mamifero unaCria);

}
```

Las clases abstractas surgen al notar que algunas clases tienen un comportamiento en común y pueden ser agrupadas bajo un mismo tipo. Pero el solo hecho de compartir código no es razón suficiente para crear una clase abstracta y hacer que esas clases hereden de la superior, sino que la razón tiene que hallarse en que la clase abstracta agrupa a todas ellas bajo un mismo concepto. Lo que se busca es reutilizar el conocimiento: la reutilización de código es una consecuencia, no la causa.

Una jerarquía correctamente diseñada forma un árbol, con Object como raíz. Las ramas se forman con clases abstractas y las hojas deben ser clases concretas (no abstractas). El objetivo es hacer que las clases padre fuercen detalles de implementación a sus clases hijas, principalmente en la forma de atributos.

Solo se deberían definir métodos con implementación y métodos abstractos. También es necesario tener presente que debemos buscar que, en vez de definir métodos abstractos, se definan métodos con una implementación por defecto. De esta manera, las clases hijas solo implementan lo que realmente desean y no se ven forzadas a implementar otros métodos.

Las clases abstractas, por definición, no pueden ser marcadas como final, ya que su propósito es que sirvan como base para otras clases.

El principio de ocultamiento de la información vale también entre clases de la misma jerarquía.

Clases anidadas

Las **clases anidadas** son aquellas que se definen en el contexto de otra clase. Existen cuatro tipos de clases anidadas: las **estáticas**, las **internas a**

la clase, las **internas a los métodos** y las **anónimas**. Exceptuando a las estáticas, las demás son para uso interno de la clase en la que se definen. Generalmente, se utilizan para modelar ciertos conceptos que están fuertemente ligados a una clase y que no tienen sentido fuera de ella, ya que su uso es en conjunto.

Clases anidadas estáticas

Conocidas como **static nested classes**, son clases definidas dentro de otra, pero independientes de la clase contenedora. Por lo tanto, pueden ser públicas y accedidas y usadas desde otras clases.

El acceso a ellas es a través de la clase contenedora. Para declararlas, definimos una clase dentro de la clase contenedora y la marcamos como estática con el modificador static:

```
public class Auto {
        public static class Llave {
                ...
}
        ...
}
```

En nuestro ejemplo, la clase interna Llave es accedida desde otras clases utilizando el nombre Auto.Llave, que deja en claro la relación que existe entre las clases Auto y Llave.

Supongamos que también definimos la clase Lancha y su correspondiente clase estática Llave. Cuando hablamos de Llave en el contexto de Lancha hacemos referencia a la clase asociada a esta; en cambio, cuando hablamos en el contexto de Auto, queremos hablar de Llave con una asociación distinta. Las clases estáticas sirven para este propósito, ya que si bien ambas tienen el mismo nombre, al tener otro contexto son dos clases totalmente diferentes.

Figura 7. Diagrama de la clase Auto y su clase anidada Llave.

Clases internas

Llamadas **inner classes**, al igual que las estáticas son clases definidas dentro de otra, aunque sus instancias siempre están ligadas a una instancia de la clase contenedora. Su definición es igual a la ya vista, sin utilizar el modificador static.

```
public class Auto {
        public class Freno {
                ...
        }
        ...
}
```

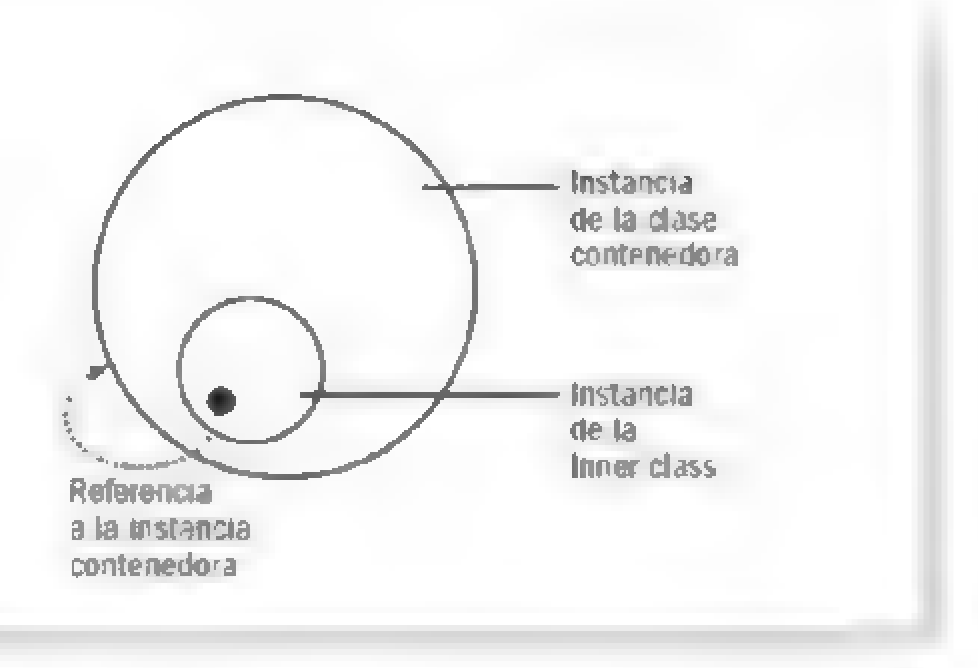


Figura 8. Relación entre una instancia de una inner class y la contenedora.

Existen tres grandes diferencias entre las static inner clases y las nested classes. La primera se halla en que las clases internas son parte de la definición de la clase (como si fuera un método de instancia), y, por lo tanto, pueden acceder a todos los elementos de esta, ya sean atributos o métodos, incluso privados. La segunda es la forma de instanciarlas. En los métodos de instancia de la clase contenedora no hay diferencia, pero si la nested class es pública, se requiere una instancia de la clase contenedora. Notemos en el ejemplo de código cómo debemos utilizar el operador new junto con la instancia de la clase contenedora:

```
public class AutoUnitTests {
        ...
        @Test public void testFrenoInnerClass() {
                Auto unAuto = new Auto();
                Auto.Freno unFreno = auto.new Freno();
        }
        ...
}
```

Por último, la tercera diferencia radica en que dentro de los métodos de instancia de la inner class tenemos acceso, no solo de referencia a la instancia correspondiente de la clase (utilizando el this), sino también a la instancia asociada de la clase contenedora.

```
public class Auto {
        class Freno {
                public void frenar() {
                        Auto.this.frenar();
// Envía frenar al objeto de tipo Auto asociado
                }
        }
        ...
}
```

Como vemos en el ejemplo, utilizamos el nombre de la clase contenedora y el this para obtener la referencia. La asociación mencionada se realiza

automáticamente cuando se instancia un objeto de la clase anidada en los métodos de instancia de la clase contenedora. Se asocia al receptor del mensaje que está siendo ejecutado.

```
public class Auto {
        class Freno {
                public void frenar() {
                        Auto.this.frenar();
                }
        }

        public Auto() {
                this.setFreno(new Freno());
// Asociación automática
}

public void frenar() {
        ...
}
...
}
```

Clases locales y clases anónimas

Las **clases locales** (*local classes*) son un tipo de clase interna que define los métodos de instancia de la clase contenedora. Están confinadas al método donde están definidas; por lo tanto, no tiene sentido

USO DE LOS STATIC IMPORTS

El uso de los **static imports** debe tener como objetivo fundamental aclarar las intenciones del programador cuando escribe código. Además, debe estar limitado solamente al uso de **static factory methods**, ya que de lo contrario estaríamos encapsulando colaboraciones interesantes en un método estático, perdiendo flexibilidad.

especificarles la visibilidad. Mantienen todas las propiedades de las inner classes con el agregado de que pueden acceder a parámetros y variables locales de método si están marcadas como final.

```
public class RelojDigital extends Reloj {
        ...
        public Tiempo ahora() {
                final Hora hora = horaActual();

                class TiempoDeRelojDigital extends Tiempo {
                        @Override
        public Hora hora() {
                                return      a;     hor
                        }
                        @Override
                        public Minutos minutos() {
                                return          RelojDigital.this.
                                             minutosActuales();
                        }
                }
                return new TiempoDeRelojDigital();
        }
        ...
}
```

Cuando especificamos el tipo de un parámetro o el tipo de retorno de un método debemos pensar siempre en la máxima abstracción que nos sirva. Esta abstracción debe ser el requerimiento mínimo de nuestro método. Utilizaremos en estos casos clases abstractas e interfaces en vez de clases concretas.

Las **clases anónimas** (*anonymous clases*) son locales, no tienen nombre y solo existen para especificar el comportamiento de la única instancia que tienen.

Estas clases son muy útiles para implementar mecanismos de **callback** (de aviso y respuesta asincrónica). No se les puede definir un constructor, pero sí tienen un bloque de código inicializador que se ejecuta después de creada la instancia. Sin embargo, se pueden utilizar los constructores de la superclase para crear la instancia anónima.

```
abstract class ObservacionSobre {
        public ObservacionSobre(Object unObjecto) { ... }
}

public class ObservacionSobreUnitTests {
        @Test public void testInicializacion() {
                Mariposa unaMariposa = new Mariposa();
                ObservacionSobre observación = new
                // Creo una instancia usando el constructor
                        ObservacionSobre(unaMariposa)        {
                        { ... } // Código inicializador
```

Primero, poseen un nombre que explica la intención del método. Esto es importante para tener código claro y explicativo Segundo, pueden no crear un objeto nuevo por cada uso Son útiles en casos donde se quieran administrar las instancias Tercero, pueden devolver objetos de un subtipo en vez del tipo en el que están definidos, y así lograr un grado de flexibilidad extra.

```
                        ... // Se pueden sobrescribir métodos
o crear nuevos; en ese sentido, es una clase normal
                };
        }
}
```

Las clases anónimas tienen las mismas características que las clases locales; por lo tanto, pueden acceder a parámetros y variables locales finales del método como a la instancia de la clase contenedora.

RESUMEN

Este capítulo presentó el modo principal de Java de catalogar comportamiento y la única forma que tiene el lenguaje para implementar código. Todo código en Java existe dentro de una clase, ya sea estático o de instancia. Hemos aprendido cómo definir atributos y métodos, qué significa la pseudovariable this y en qué se diferencia de super. También vimos la diferencia entre el alcance de instancia y el estático, y los motivos para evitar el uso de este último. Por último, empezamos a ver cómo se hereda comportamiento de otras clases y cómo formar una jerarquía de tipos.

Actividades

TEST DE AUTOEVALUACIÓN

1 ¿De cuántas clases se puede heredar?

2 ¿Cuántas interfaces puede implementar una clase?

3 ¿Qué es un **atributo**?

4 ¿Qué son los **getters** y los **setters**?

5 ¿Qué es this? ¿Qué es super?

EJERCICIOS PRÁCTICOS

1 Cree una clase, defina un atributo y sus métodos de acceso.

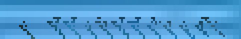
2 Redefina el método toString (definido en Object)

3 Agregue un método a la clase y sobrecargue el método añadiendo nuevos parámetros.

4 Cree una subclase y sobrescriba un método de la clase padre utilizando super.

5 Implemente una grilla cuadrada alternativa a la original.

Si tiene alguna consulta técnica relacionada con el contenido, puede contactarse con nuestros expertos: **profesor@redusers.com**.



Interfaces y enumeraciones

Como mencionamos en capítulos anteriores, existen otros mecanismos para reutilizar código sin caer en las trampas de la herencia múltiple. Uno de estos mecanismos son las interfaces. Al terminar este tema, conoceremos algunas estructuras de código que nos serán útiles para realizar el desarrollo de nuestras aplicaciones: las enumeraciones.

Interfaces

Una interfaz es solamente la declaración de los mensajes que sabe responder un determinado tipo, es decir, la declaración del protocolo aceptado por los objetos de ese tipo. Las interfaces no suplantan a las clases, sino que las complementan. Una clase, que es protocolo más comportamiento, implementa una o varias interfaces, que solo son protocolo.

Para implementar una interfaz, una clase tiene que poder responder los mensajes declarados en ella, ya sea implementándolos o definiéndolos como abstractos. Las interfaces, al igual que las clases abstractas, no pueden tener instancias directamente. Es posible hacer que una interfaz extienda otra, requiriendo que la clase que implementa tal interfaz implemente también la que extiende.

Las interfaces declaran la firma de los mensajes y no contienen implementación alguna. De esta forma, no sirven para reutilizar código directamente, pero sí para otras funciones más importantes.

Recordemos que, en el capítulo anterior, comentamos que una característica deseable de un sistema era el bajo acoplamiento. Podemos reducir el acoplamiento disminuyendo la cantidad de clases de las cuales dependemos, o haciendo que las dependencias sean con tipos de protocolo reducido. Los protocolos reducidos no solo alivianan las dependencias, sino que también hacen que un determinado tipo sea más entendible, ya que hay que comprender menos mensajes para saber cómo responde un objeto.

Una interfaz por sí sola no puede tener instancias, sino que depende de una clase que la implemente. Podemos probarlo tratando de instanciar una y ver qué nos responde el compilador o Eclipse. Por otro lado, pensemos lo siguiente: si una interfaz solo define protocolo, firmas de métodos y mensajes que debe entender, ¿qué es lo que ejecutaría una instancia de tal interfaz?

Las interfaces son artefactos que permiten agrupar mensajes relacionados y separarlos de otros mensajes no relevantes para una determinada funcionalidad. Una clase que implementa varias interfaces tiene la capacidad de responder a varias funcionalidades, pero los clientes de dicha clase no tienen por qué saber todo lo que ella puede hacer (solo deben conocer la interfaz que les interesa). Así, se evita engrosar las dependencias entre los elementos, ya que los clientes son independientes de los cambios que puedan ocurrir en los otros mensajes.

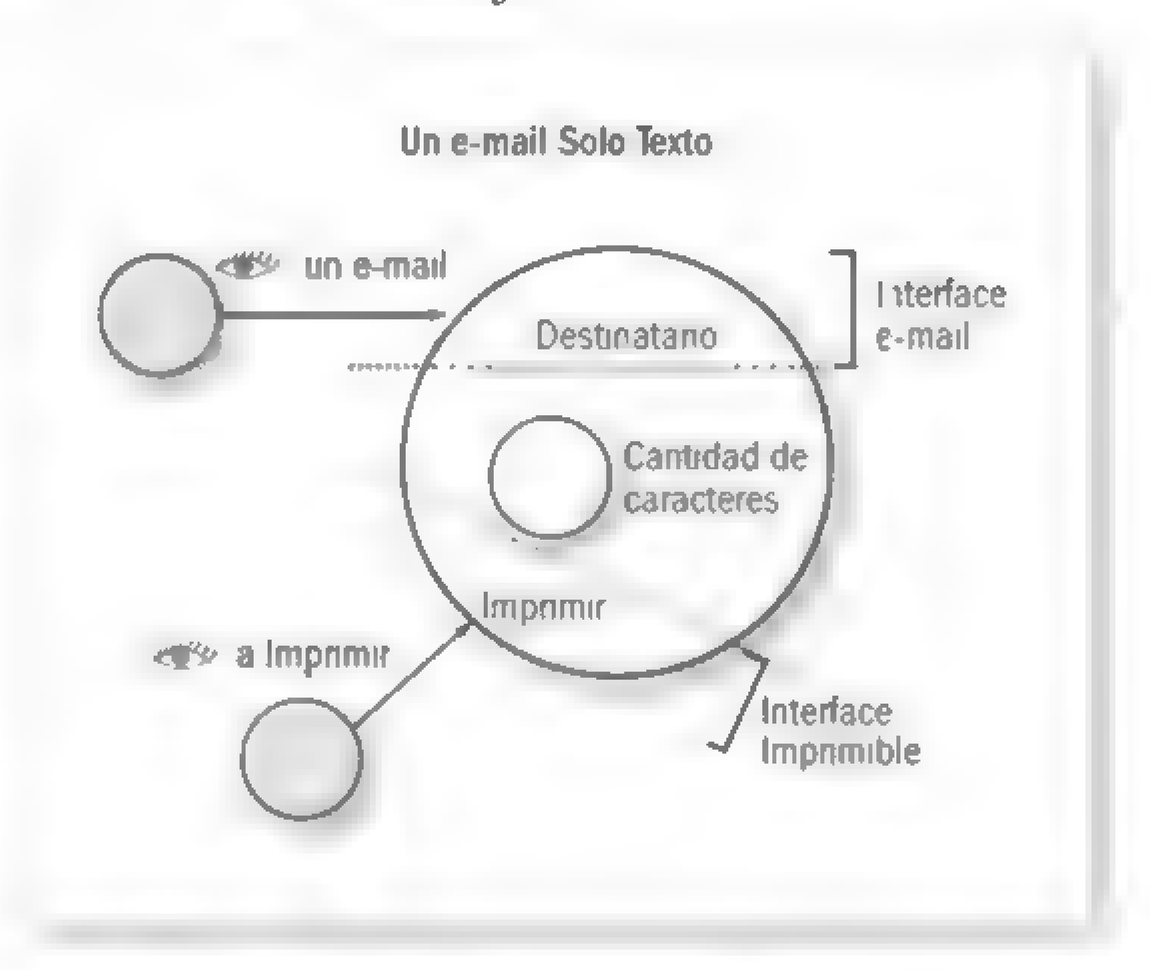


Figura 1. Las interfaces permiten restringir el protocolo usado por un objeto cliente a los mensajes necesarios solamente para una funcionalidad.

Para Java, las interfaces también son tipos como lo son las clases; por lo tanto, los objetos pertenecen al tipo determinado por su clase y a los tipos establecidos por las interfaces que implementan. Estos forman jerarquías que no siguen la línea de las clases, sino que hay una jerarquía por interfaz y por las clases que las implementan. Podemos decir que se trata de una relación de tipos que es transversal a la clasificación.

Uso

Para definir una interfaz en Java utilizamos la palabra interface en vez de class. Esta acepta los distintos modificadores de visibilidad:

```
public interface Mensaje {
        ...
}
```

La definición del protocolo se realiza definiendo las firmas de los métodos, sin cuerpo (sin código), de igual forma que cuando definimos un método abstracto. Todos los métodos indicados en una interfaz son abstractos, por lo que utilizar este modificador es redundante. Ocurre algo similar con la visibilidad de los métodos: todos son públicos.

```
public interface Mensaje {
        Destinatario destinatario();
        ...
}
```

Eclipse nos facilita la tarea de crear una interfaz desde cero. Para ello, deberemos hacer clic en el menú File o sobre el botón de New en la barra de herramientas; también podemos hacer clic con el botón secundario del

INTERFACES CON CONSTANTES

Si bien es posible definir constantes en las interfaces, no es una práctica del todo recomendable. Generalmente, las interfaces que son puramente constantes pueden ser reemplazadas por **enumeraciones**, o bien moviendo las constantes a alguna clase abstracta o a la clase que las utiliza. Además, no es posible sobrescribir una constante ya que no se adaptan bien al cambio

mouse y seleccionar New.../Interface. Luego elegiremos la carpeta donde se encuentran los archivos fuente del proyecto y el paquete de destino, ingresaremos el nombre de la interfaz y seleccionaremos posibles interfaces a extender. Para terminar, presionaremos Finish.

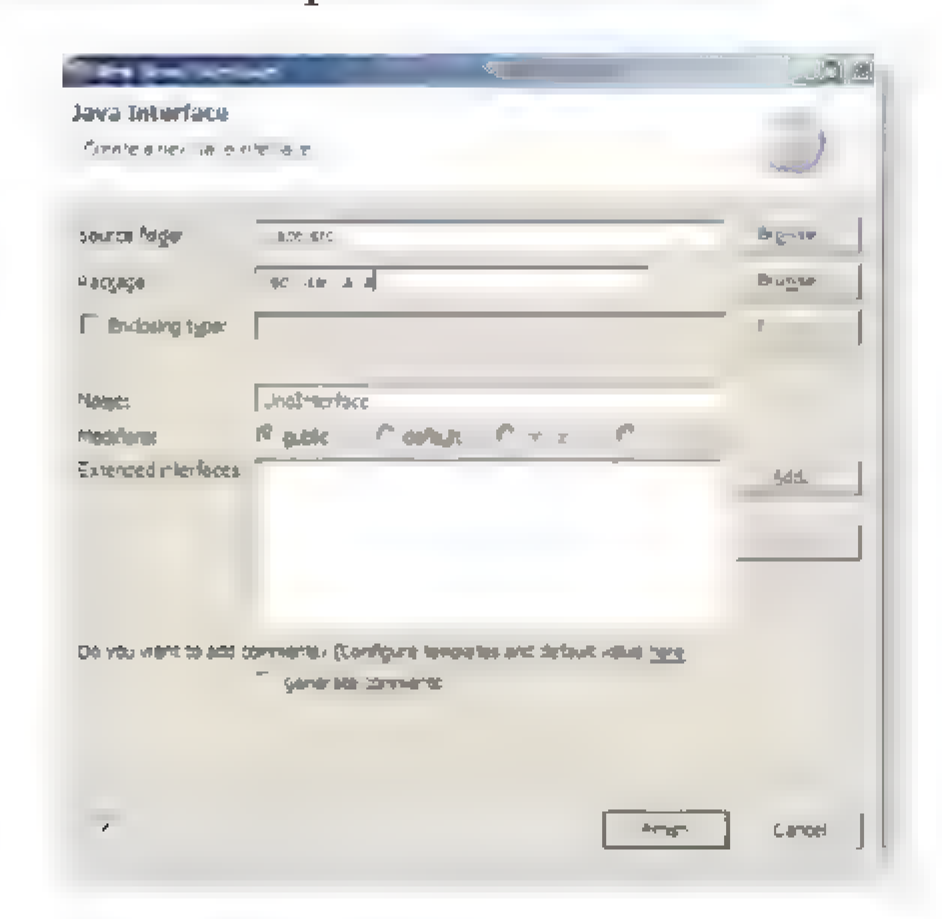


Figura 2. Pantalla de creación de interfaces.

En las interfaces no se pueden definir atributos de instancia sino solo métodos. Tampoco se pueden definir métodos estáticos, ya que lo único que se puede definir, además del protocolo, es una constante. Las constantes son atributos públicos y estáticos, generalmente se escriben en mayúsculas y se las marca como final.

```
public interface MiInterfaceConConstantes {
        static final double PI = 3.14d;
        ...
}
```

Al igual que en el caso del protocolo, el atributo public es redundante, ya que todo en una interfaz es público.

Como hemos afirmado, una interfaz no hereda de otra (o de muchas otras), sino que extiende. Cuando una interfaz extiende a otra, amplía el protocolo definido por esta con nuevos métodos.

```
public interface Email extends Mensaje {
        ... // Defino nuevos mensajes
}
```

Es necesario tener en cuenta que, antes de que aparecieran las anotaciones en Java, uno de los usos de las interfaces era el de marcar a las clases con información.

Un ejemplo que se encuentra en el SDK es la **interfaz serializable**. Estas interfaces están absolutamente vacías y solo aportan por su presencia. Hoy lo más conveniente es utilizar anotaciones para este propósito, ya que son más poderosas. Si bien se utiliza la misma palabra, extends, para extender una interfaz y para heredar de una clase, la semántica es bien distinta. En un caso extendemos un protocolo, mientras que en el otro se crea una jerarquía de clases sin necesidad de extender protocolo. Cuando una clase quiere implementar una o varias interfaces, lo hace mediante la palabra implements, seguida de los nombres de las interfaces que desea implementar separados por coma:

```
public class EmailSoloTexto implements Email {
@Override    Implemento los mensajes de la interfaz
public Destinatario destinatario() {
        ...
}
...
}
```

Al igual que cuando implementamos un método abstracto o sobrescribimos un método de alguna clase padre, conviene anotar el método que

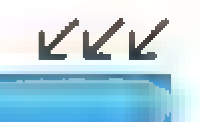
implementa un mensaje de una interfaz con @Override. También debemos acordarnos de agregar el modificador de visibilidad public para no romper el contrato con la interfaz, ya que tiene que ser público.

Es posible utilizar una interfaz para definir una clase anónima en un método. El mecanismo es el mismo que vimos anteriormente:

```
...
Mensaje mensaje = new Mensaje() {
@Override
public Destinatario destinatario() {
        ...
}
        ...
};
...
```

Clases abstractas versus interfaces

Es difícil, al principio, determinar cuándo es necesario o conveniente crear una interfaz o una clase abstracta. Recordemos que ambos artefactos tienen propósitos distintos: las clases abstractas sirven para establecer una jerarquía de tipos y proveer un comportamiento básico, y las interfaces son definiciones de protocolos, es decir, determinan el comportamiento de un tipo.



Una gran ventaja de usar interfaces en vez de referenciar directamente clases se encuentra en que, a la hora de escribir nuestros test, estas dependencias son mucho más fáciles de proveer. Solamente es necesario crear un simple objeto que implemente la interfaz, algo que podemos hacer incluso en el mismo método con clases anónimas.

Por otro lado, las clases abstractas pueden incluir código, definir atributos, tener métodos estáticos, etcétera; en cambio, las interfaces son solo declaraciones de mensajes.

Por este motivo las interfaces son mucho menos restrictivas que las clases abstractas, ya que no fuerzan nada más que lo necesario: solo el protocolo. Por ejemplo, una clase abstracta podría definir un atributo de instancia; esto es en sí una restricción de implementación, ya que todas las subclases tienen como peso ese atributo (incluso si no lo usan).

Además, la herencia es una clase que se puede utilizar una sola vez. Solo podemos heredar de una clase e implementar todas las interfaces necesarias. Nuestra clase es, entonces, polifacética, ya que puede servir a muchos más clientes.

Generalmente, si nuestra clase abstracta solo define métodos, lo mejor es crear una interfaz y reemplazarla. Así, ganaremos un poco de flexibilidad y dejaremos la clase abstracta para cuando queramos reutilizar comportamiento (código) en común entre varias clases similares.

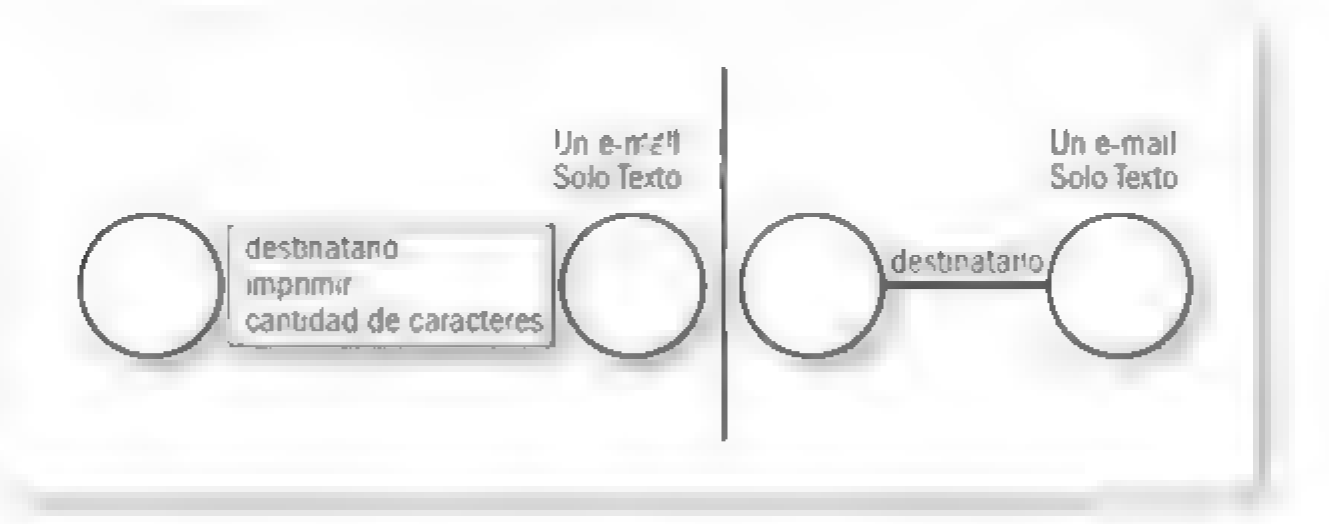


Figura 3. Las interfaces permiten reducir el acoplamiento entre dos objetos.

Cuando estamos programando una clase y sabemos que vamos a necesitar ciertos colaboradores que todavía no existen, podemos crear una interfaz. Nuestra clase compilará habiendo creado los artefactos mínimos para esto

Si queremos definir una interfaz a partir de una clase existente y que esta la implemente, contamos con una funcionalidad de Eclipse para eso. Para extraer una interfaz, debemos presionar el botón derecho del mouse sobre

el código de una clase y seleccionar la opción Refactor/Extract Interface … . Para continuar, ingresaremos un nombre para la nueva interfaz, elegiremos los métodos de la clase que pertenecerán a esta y luego haremos clic en OK.

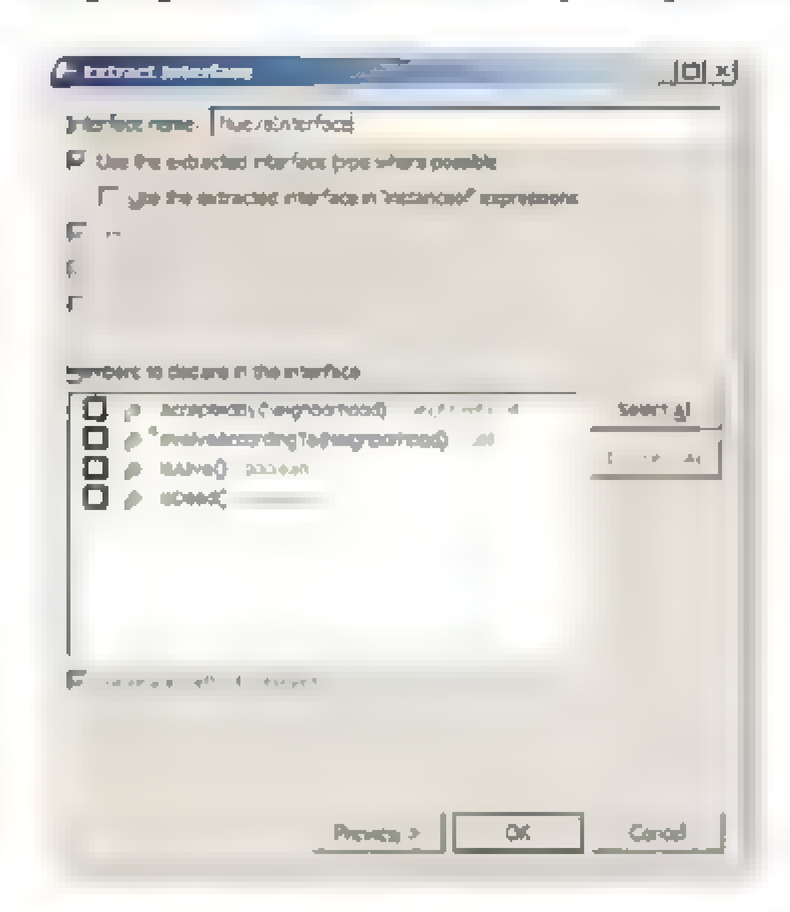


Figura 4. Pantalla de extracción de interfaz a partir de una clase.

Enumeraciones

Las enumeraciones son la solución de Java al viejo problema de utilizar números enteros o strings para representar un conjunto de elementos acotado y bien definido, donde cada uno tiene un nombre. Estas enumeraciones son seguras porque tienen un tipo explícitamente definido para ellas, sin poder intercambiarse con otros tipos y cometer, en consecuencia, errores. Al mismo tiempo, las enumeraciones están acompañadas de dos clases auxiliares para poder manejarlas en conjunto de forma fácil y eficiente.

Es común que ciertos dominios que se nos presentan a la hora de modelar tengan elementos distinguidos. Imaginemos una baraja de cartas, con cuatro clases de naipes (corazón, trébol, pica y diamante), numerados del dos al nueve, seguidos de las figuras J (sota), Q (reina), K (rey) y A (as).

Aquí tenemos dos ejemplos de enumeraciones: los cuatro palos y las diez cartas. El palo trébol es un elemento distinguido del conjunto de palos, del mismo modo que el número cinco lo es del conjunto de la numeración de las cartas. Las enumeraciones permiten, entonces, definir un conjunto de elementos distinguidos. En las versiones anteriores de Java se definían utilizando constantes de tipo numérico o de cadena de caracteres

```
public class Palo {
        public static final String PICA = "♠";
        public static final String CORAZON = "♥";
public static final String DIAMANTE = "♦";
public static final String TREBOL = "♣";
}
```

Figura 5. Diagrama de una enumeración con el estilo viejo

El problema de este tipo de enumeraciones es que PICA no es de tipo Palo y, por lo tanto, en todos los lados donde queramos usar un palo aparecerá el tipo String. Imaginen un método que está esperando un tipo Palo y recibe la cadena "Hola mundo!" : seguramente, rompería el programa.

Una solución para estos casos es utilizar el tipo Palo y hacer que cada palo sea una instancia. Para asegurar que no se creen más palos que los determinados, se marca como privado el constructor:

```
public class Palo {
        public static final Palo PICA = new Palo("♠");
public static final Palo CORAZON = new Palo("  ♥");
public static final Palo DIAMANTE = new Palo("   ♦");
public static final Palo TREBOL = new Palo("   ♣");

        private String símbolo;

        private Palo(String símbolo) {...}
        ...
}
```

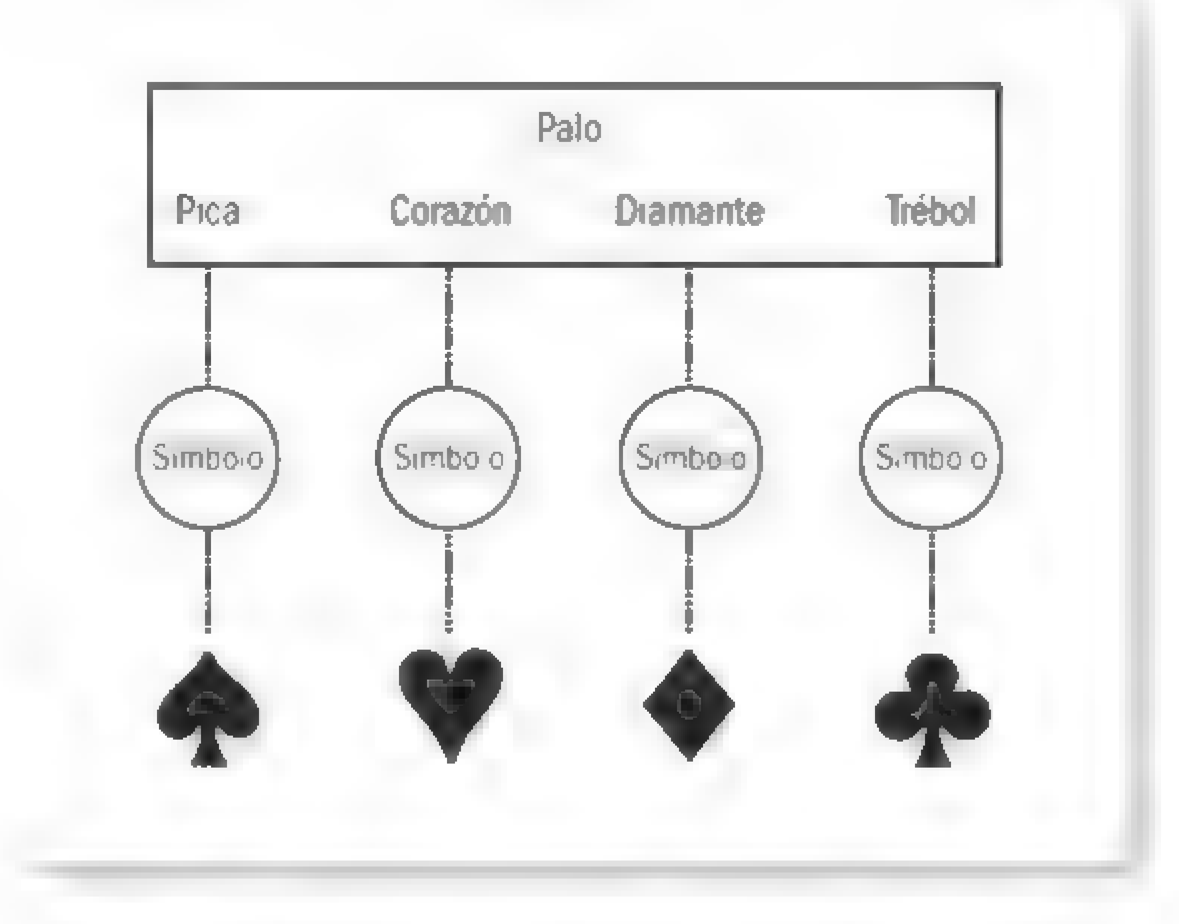


Figura 6. Diagrama de una enumeración utilizando objetos que modelan correctamente el concepto.

Estas construcciones representan una gran mejora frente a las otras enumeraciones, ya que hay un tipo determinado para representar a los palos e instancias de ese tipo.

Las enumeraciones modernas en Java son un artefacto sintáctico que produce el mismo resultado que la construcción anterior, pero sin escribir tanto. El lenguaje tiene el concepto de enumeración, es decir que no se trata de algo que solamente está en la mente del programador.

Uso

A partir de la versión 1.5, definir una enumeración en Java se consigue mediante el uso de la palabra clave enum, en lugar de class. Luego, en el cuerpo de la enumeración, se definen los elementos distinguidos:

```
public class DiaDeLaSemana {
        DOMINGO, LUNES, MARTES, MIERCOLES,
JUEVES,VIERNES, SABADO
}
```

Cada día de la semana así definido es un objeto instancia de DiaDeLaSemana.

Eclipse facilita la creación de enumeraciones al proveer una ventana para esta tarea. Debemos dirigimos al menú File o al botón New en la barra

NIVEL DE ABSTRACCIÓN

Cuando diseñamos interfaces, al igual que cuando lo hacemos con las clases, debemos ser lo mas abstractos posible. Eso significa que en una interfaz debemos definir el mínimo conjunto de mensajes que tengan sentido, que esten relacionados bajo un concepto, y evitar agregar metodos extraños. En esos casos, es mejor extender la interfaz y agregarlos en la nueva.

de herramientas, elegir la carpeta donde se encuentran los archivos fuente del proyecto y seleccionar el paquete destino. Luego, escribiremos el nombre de la enumeración y elegiremos las posibles interfaces que implementará la enumeración. Para terminar, presionaremos Finish

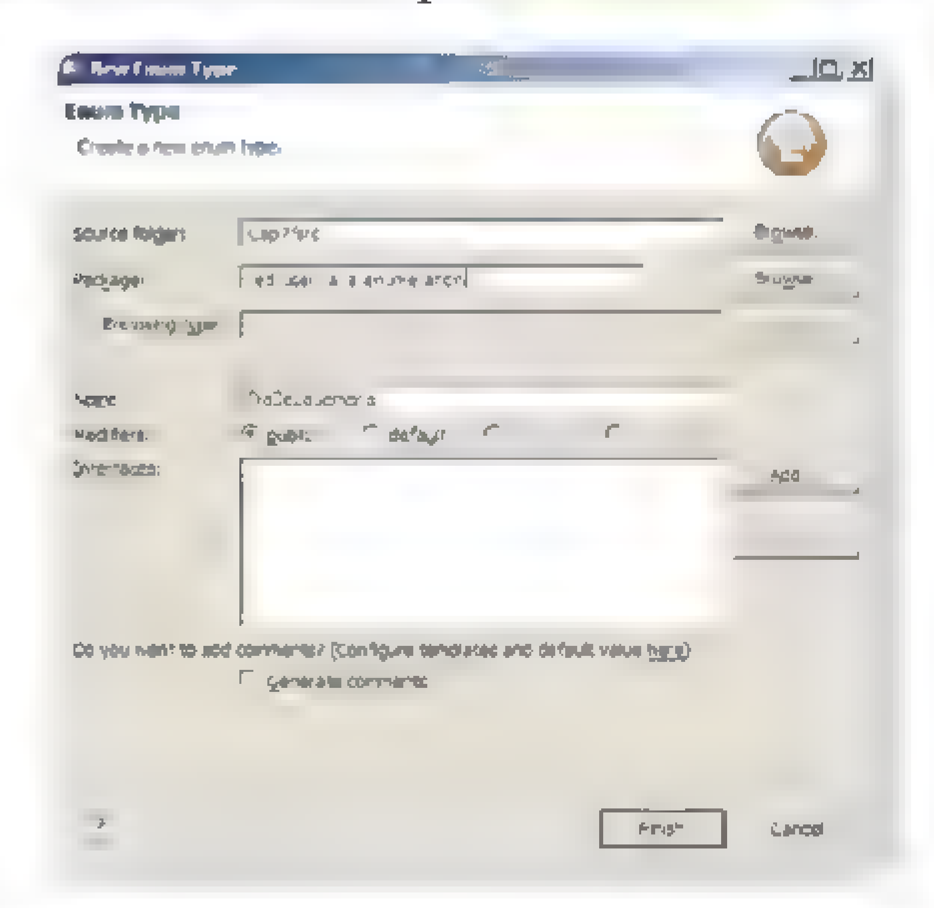


Figura 7. Vista de la ventana de creación de enumeraciones en Eclipse.

Al ser también clases, las enumeraciones pueden definir sus propios métodos y atributos, al igual que los constructores.

```
public enum Palo {
        PICA("♠"),
        CORAZON("♥"),
        DIAMANTE("♦"),
        TREBOL("♣");

        private String simbolo;

        Palo(String simbolo) {
                this.simbolo = simbolo;
```

```
        }

        public String simbolo() {
                return simbolo;
        }

        @Override
        public String toString() {
                return simbolo();
        }
}
```

Los constructores en las enumeraciones solo pueden ser privados o tener privacidad default. No es posible hacerlos públicos, porque entonces se podrían crear otros elementos con ellos. Como se ve en el ejemplo, las enumeraciones también son clases, en consecuencia, es posible hacer que implementen interfaces:

```
public interface Palo {
        String simbolo();
}
```

Dado que generalmente las **enumeraciones** representan entes constantes e inmutables, es una convención aceptada que los nombres de los elementos estén totalmente escritos en mayúsculas. Esto facilita, en la lectura, reconocer qué elementos son enumeraciones y cuáles no, sin tener que navegar todo el código.

Si queremos implementarla para los palos de la baraja inglesa, haremos lo siguiente:

```
public enum PaloIngles implements Palo {
        PICA("♠"),
        CORAZON("♥"),
        DIAMANTE("♦"),
        TREBOL("♣");

        ...
}
```

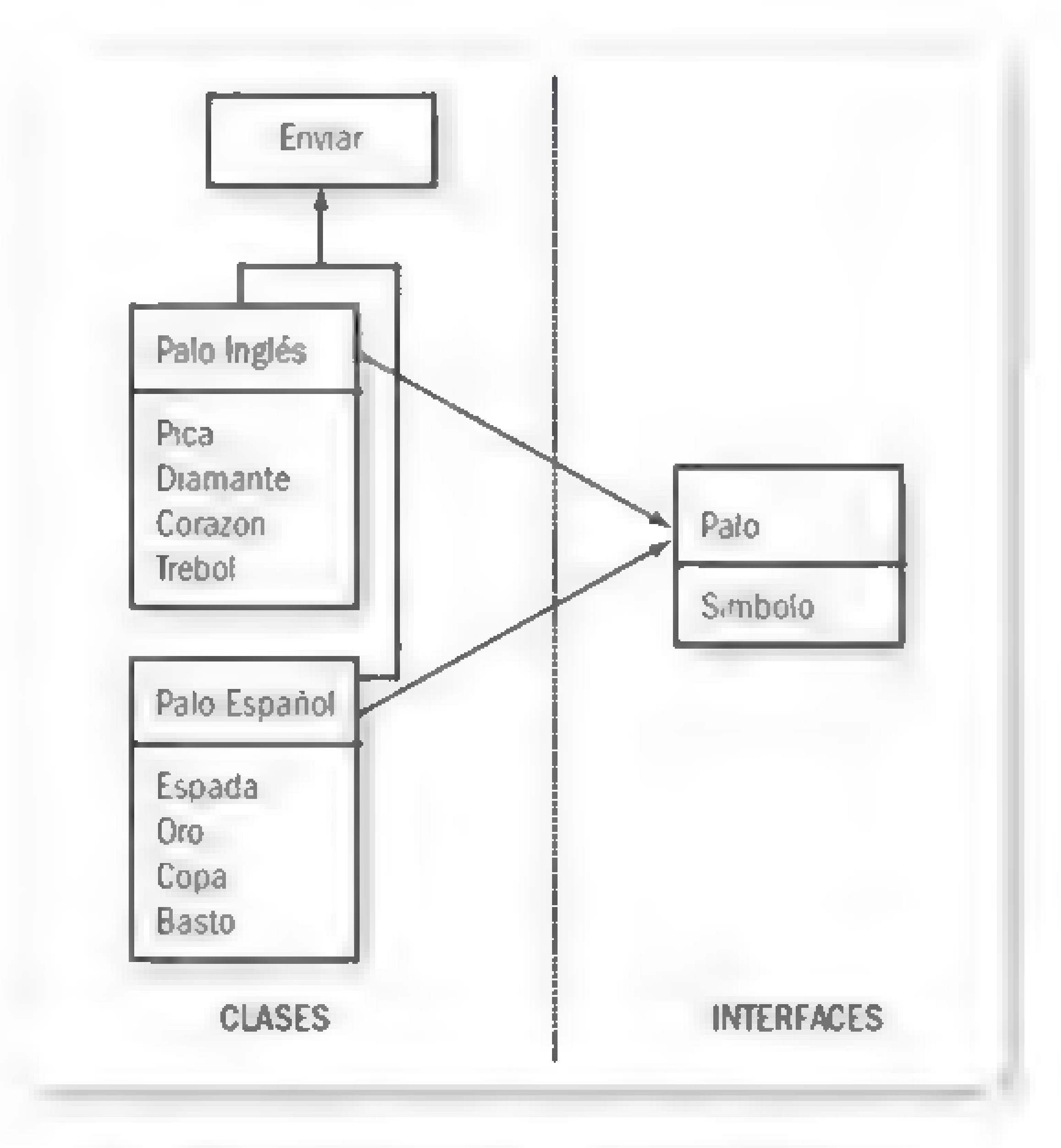


Figura 8. Es posible implementar interfaces en las enumeraciones para formar relaciones de tipo.

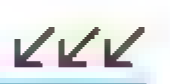
Las enumeraciones heredan automáticamente de la clase Enum de Java y no se les puede incorporar ninguna extensión, ni siquiera de otra enumeración, ya que son **clases finales**. Por heredar de Enum, los elementos de una enumeración tienen varios métodos interesantes. El primer método es name(), que devuelve el nombre con el que identificamos el elemento:

```
//True
assertEquals(Palo.PICA.name(), "PICA");
```

El otro método es ordinal(), que permite conocer el orden en el cual fue definido el elemento, comenzando por el cero:

```
//True
assertEquals(Palo.DIAMANTE.ordinal(), 2);
```

La clase Enum también provee algunos métodos estáticos útiles para manejar las enumeraciones. Uno es el método values(), que retorna un array con todos los elementos de la enumeración, en el orden en que fueron definidos. El segundo es valueOf(String), que permite obtener el elemento que se llama como el string entregado como argumento. Si el nombre entregado no coincide, arroja una IllegalArgumentException



Cuando trabajamos con elementos distinguidos, es conveniente que importemos estáticamente la enumeración a la cual pertenecen. De este modo, la escritura del código que los utiliza es mucho más sencilla, como también lo es su lectura, ya que no hay que estar leyendo una y otra vez el nombre de la enumeración como prefijo.

```
//True
assertArrayEquals(Palo.values(), new Palo[] { PICA,
CORAZON, DIAMANTE, TREBOL });
//True
assertEquals(TREBOL, Palo.valueOf("TREBOL"));
//Arroja una IllegalArgumentException
Palo.valueOf("trebol");
```

Los elementos de una enumeración pueden están ordenados según el orden de definición; por lo tanto, se los puede comparar con el método compareTo(Enum) de la interfaz Comparable que Enum implementa.

```
public enum DEFCON {
        FADE_OUT(5),
        DOUBLE_TAKE(4),
        ROUND_HOUSE(3),
        FAST_PACE(2),
        COCKED_PISTOL(1);

        ...
}
```

IGUALDAD E IDENTIDAD

Los elementos de una enumeración son **entidades únicas** y, por lo tanto, siempre que usemos alguno de ellos estaremos usando la misma instancia. Entonces, utilizar la igualdad (método equals) o utilizar la identidad (operador ==) es indistinto, ya que arrojan los mismos resultados.

En el ejemplo, se observa una enumeración para los niveles de **DEFCON** (niveles de defensa de los Estados Unidos), desde el nivel **DEFCON 5** que es el estadio normal, hasta **DEFCON 1** que significa guerra. **DEFCON 5** es entonces el menor nivel de alerta, lo que se refleja en su orden de definición.

```
//True
assertTrue(FADE_OUT.compareTo(COCKED_PISTOL) < 0);
```

Debemos evitar el uso del orden de definición como información adicional, tal como muestra el siguiente ejemplo, ya que puede cambiar y afectar código escrito dependiente:

```
public enum Piso {
PB, PRIMERO, SEGUNDO, TERCERO;
        public int numero() { return ordinal(); }
}
```

Para guardar este tipo de información, es recomendable utilizar atributos:

```
public enum Piso {
PB(0), PRIMERO(1), SEGUNDO(2), TERCERO(3);
private int numero;

Piso(int numero) { this.numero = numero; }

        public int numero() { return numero; }
}
```

Las enumeraciones son soportadas por la estructura switch de manera natural, aunque esta no sea la mejor forma de realizar una determinada acción, sobre la base del elemento de la enumeración.

```
enum Operación {SUMA, RESTA, MULTIPLICACION, DIVISION}
...
switch(unaOperacion) {
        case SUMA: a + b; break;
        case RESTA: a – b; break;
        case MULTIPLICAION: a * b; break;
        case DIVISION: a / b; break;
        default: throw new AssertionError("Operación desconocida");
}
```

Si fuéramos a agregar un nuevo elemento a la enumeración, tendríamos que buscar en todos los lugares donde usamos el switch y agregarlo ahí.

Una gran característica de estas enumeraciones es que podemos especificar comportamiento por instancia, como si estuviéramos utilizando clases anónimas para definirlas. Así, aprovechamos el hecho de que son objetos y podemos hacer uso del **polimorfismo**.

```
public enum Operacion {
        SUMA {
                @Override
                public double aplicadaA(double unOperador,
double otroOperador) {
                        return unOperador + otroOperador;
                }
        },
        RESTA {
                @Override
                public double aplicadaA(double unOperador,
double otroOperador) {
                        return unOperador - otroOperador;
                |
        },
        ...;

        public abstract double aplicadaA(
                double unOperador, double otroOperador);
}
```

Existen dos clases fuertemente relacionadas al uso de las enumeraciones: EnumSet y EnumMap. Estas clases están altamente especializadas y optimizadas para trabajar con enumeraciones y vienen a reemplazar antiguas prácticas, como la de utilizar máscaras de bits.

La clase EnumSet es un Set para trabajar exclusivamente con elementos de una enumeración en particular y posee varios métodos estáticos que facilitan la creación de estos conjuntos. El primero es allOf(Class), que permite obtener todos los elementos de una enumeración como un conjunto:

```
Set<Palo> palos = EnumSet.allOf(Palo.class);
//True
assertEquals(4, palos.size());
```

También existe el método noneOf(Class), que devuelve un set vacío del tipo de enumeración pasado como parámetro:

```
Set<Palo> vacio = EnumSet.noneOf(Palo.class);
//True
assertTrue(vacio.isEmpty());
```

Cuando nuestros elementos distinguidos tienen atributos asociados a ellos, es conveniente proveer uno o varios métodos que, dado alguno de los atributos, devuelva el elemento correspondiente. Funcionarían de forma similar a la que trabaja valueOf, que, dado el nombre, devuelve el elemento (podemos sobrecargar valueOf siempre que sea posible).

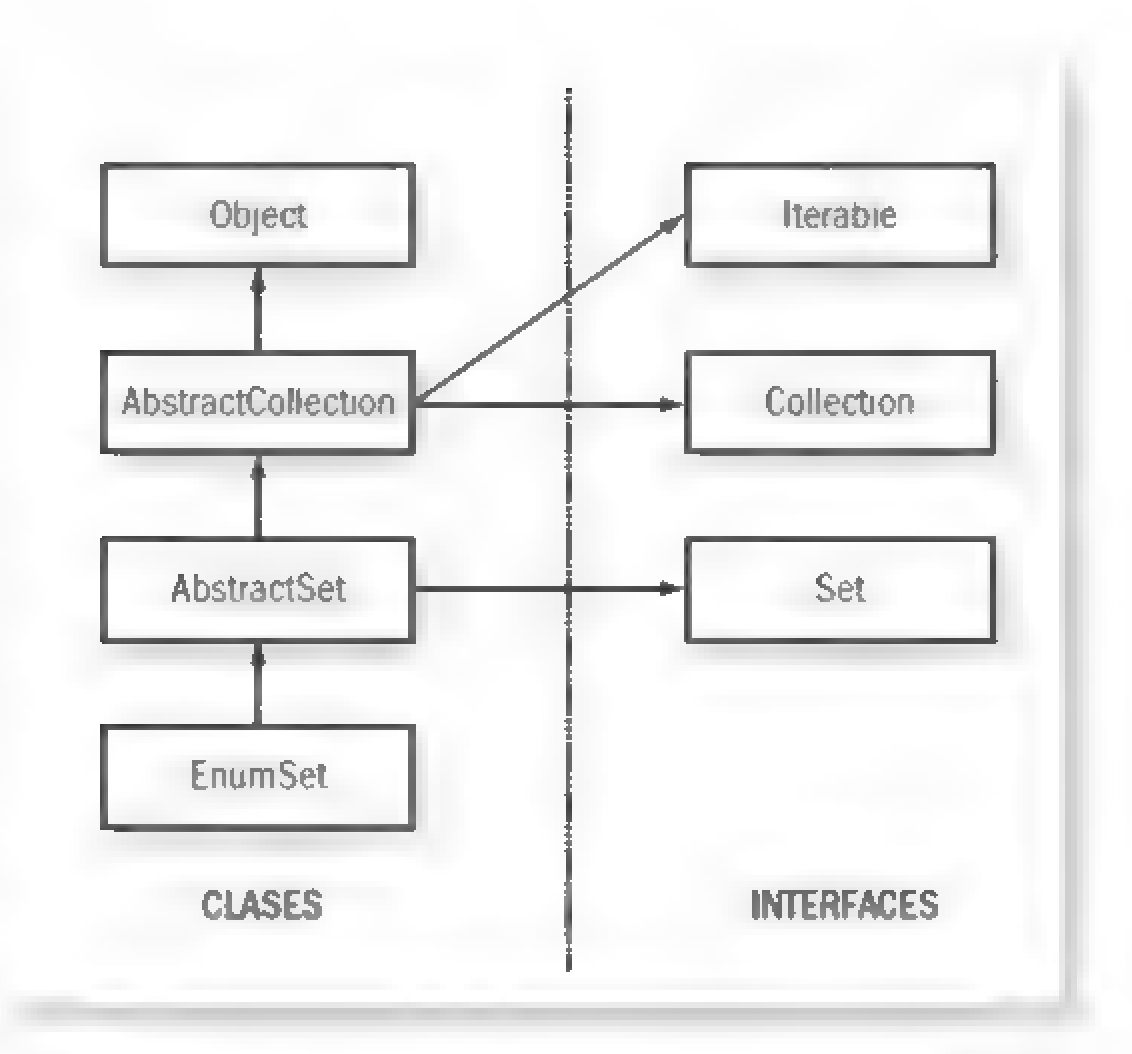


Figura 9. Jerarquía de la clase EnumSet.

Otro método importante es of(Enum ...), que permite crear un conjunto con los elementos especificados:

```
EnumSet<Palo> rojos = EnumSet.of(CORAZON, DIAMANTE);
```

Las enumeraciones permiten definir conjuntos de **singleton**, elementos conocidos bien definidos. Por este motivo, no deben ser utilizados de más sino solamente cuando tengan sentido en el dominio del problema, principalmente porque no son extensibles, ya que son clases finales y objetos globales que generan mucha dependencia.

El método complementario a of es complementOf(Enum . .) y sirve, precisamente, para conseguir el conjunto complemento del especificado:

```
EnumSet<Palo> negros   EnumSet.complementOf(rojos);
//True
assertTrue(negros.equals(EnumSet.of(PICA, TREBOL)));
```

Por último, EnumSet provee un método para obtener todos los elementos de una enumeración dentro de un rango. Se basa en el orden de definición:

```
//EnumSet con LUNES. MARTES. MIÉRCOLES. JUEVES y VIERNES
EnumSet<DiaDeLaSemana> diasHabiles =
EnumSet.range(LUNES,VIERNES);
//True
assertTrue(EnumSet.complementOf(diasHabiles).
equals(EnumSet.of(SABADO, DOMINGO)));
```

El EnumSet es muy útil cuando se busca combinar elementos de enumeraciones; por ejemplo, si se quiere equipar un auto con distintos componentes:

```
unAuto.equipadoCon(EnumSet.of(AIRE_ACONDICIONADO,
STEREO_MP3, GPS));
```

La clase EnumMap se utiliza cuando se quiere relacionar cierta información con los elementos de una enumeración. En EnumMap, la clave de búsqueda es el elemento distinguido. Por ejemplo, el plato del dia de un restaurante:

```
EnumMap<DiaDeLaSemana, String> platosDelDia =
new EnumMap<DiaDeLaSemana, String>
(DiaDeLaSemana.class);
platosDelDia.put(LUNES, "milanesa");
platosDelDia.put(MARTES, "pasta");
platosDelDia.put(MIERCOLES, "pescado");
platosDelDia.put(JUEVES, "sopa");
platosDelDia.put(VIERNES, "asado");
```

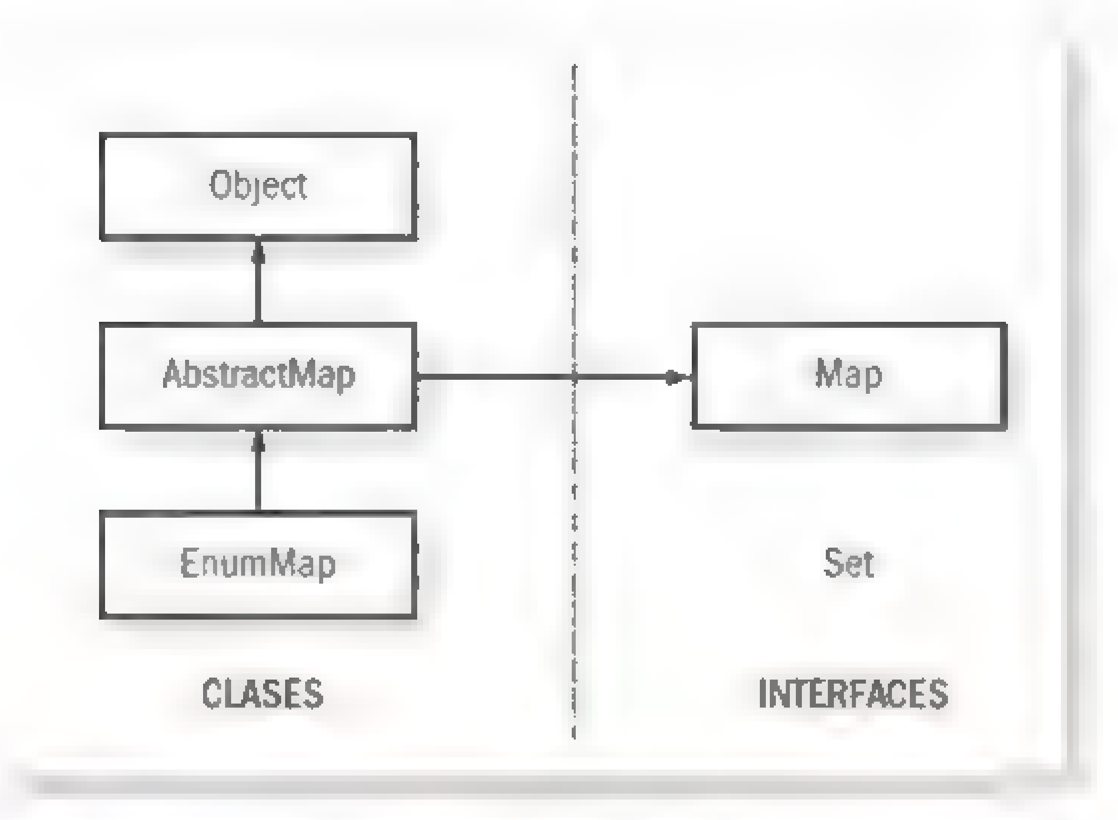
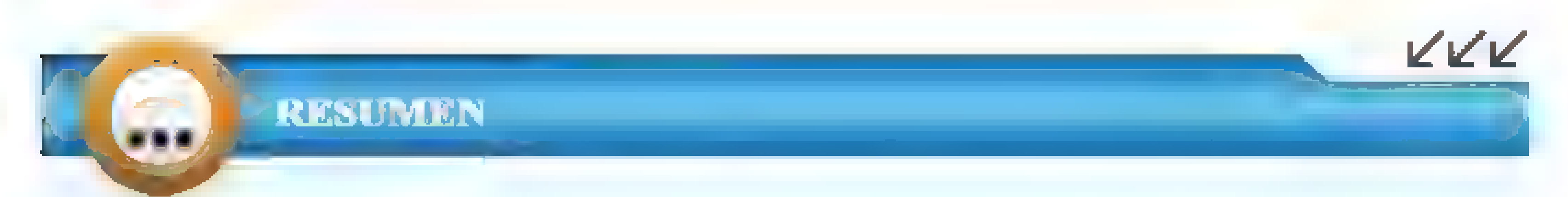


Figura 10. Jerarquía de la clase EnumMap.



A lo largo de este capítulo hemos conocido técnicas para la utilización tanto de interfaces (un poderoso mecanismo de polimorfismo para un lenguaje estáticamente tipado como Java) como de enumeraciones.

Actividades

TEST DE AUTOEVALUACIÓN

1 ¿Qué es una **interfaz**? ¿Qué elementos puede contener?

2 ¿Puedo declarar métodos privados en una interfaz?

3 ¿Cuántas interfaces puede implementar una clase?

4 Enumere algunas diferencias entre las interfaces y las **clases abstractas**.

5 ¿Qué es una **enumeración**?

EJERCICIOS PRÁCTICOS

1 Utilice el proyecto Juego de la Vida y cambie la clase Default a una interfaz.

2 Mueva el método toNull a una clase nueva.

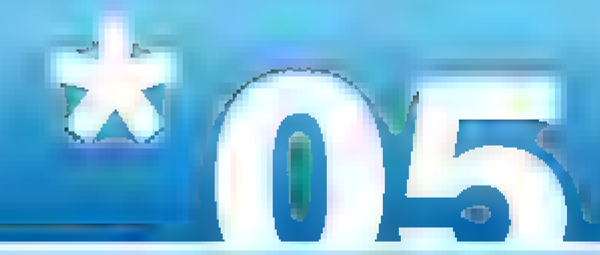
3 Remueva el método toNull y cree una clase propia para ese Default.

4 Cambie la jerarquía de Neighborhood para que sea una interfaz, una clase abstracta y el resto de las clases existentes.

5 ¿Podría Cell ser una interfaz? Haga la prueba.

PROFESOR EN LÍNEA

Si tiene alguna consulta técnica relacionada con el contenido, puede contactarse con nuestros expertos: **profesor@redusers.com**.



Excepciones y genéricos

Cuando nuestro programa está siendo ejecutado y ocurre alguna situación inesperada, el sistema lo notifica mediante eventos denominados excepciones. Las excepciones son la forma de avisar y detectar errores o acontecimientos extraños en la ejecución.

Excepciones

Una **excepción** es un evento no esperado que ocurre durante la ejecución de un programa y que interrumpe su flujo normal. Cuando ocurre un error durante la ejecución de un método, este crea un objeto con información sobre las causas del error en el contexto de la ejecución, que es pasado al sistema (a la máquina virtual) para que lo trate.

Esto se conoce como **lanzar una excepción**. Cuando lanzamos una excepción, el sistema trata de encontrar algún método en la cadena de llamados que puede manejarla. Para entender cómo funciona esto, primero tenemos que comprender cómo funcionan las llamadas a **métodos**.

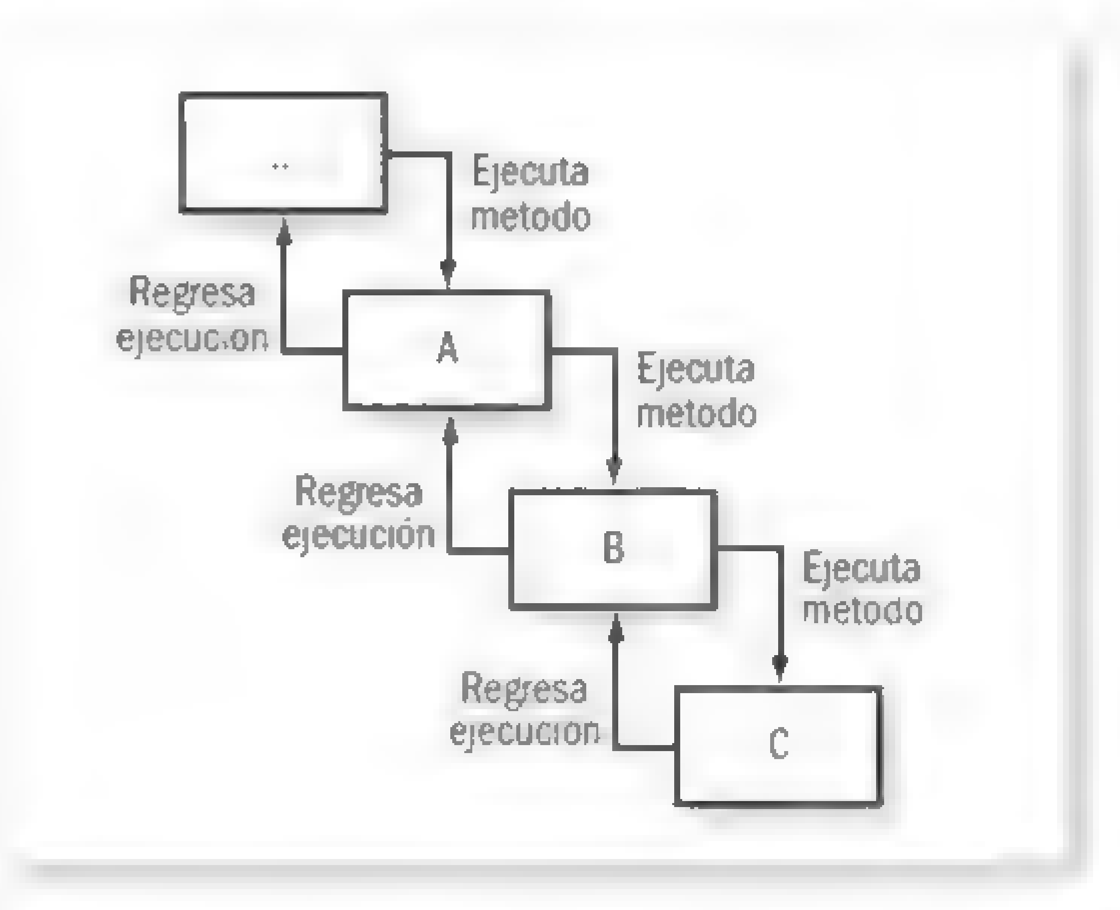


Figura 1. Un **stack** está formado por invocaciones a métodos; además, crece y decrece a medida que se termina la ejecución particular de un método

Cuando el sistema ejecuta un método asociado a un envio de mensaje (o un método estático), crea una estructura en la memoria con la información relacionada, el método y los argumentos, entre otros datos. Estas estructuras, llamadas **registros de activación** (*activation records*).

se van enlazando a medida que llamamos de un método a otro. De esta forma, cuando un método finaliza su ejecución, se continúa con la ejecución del método anterior en la cadena.

A esta cadena se la conoce como **pila de llamadas** (*call stack*), y proviene de la representación en la memoria de los registros de activación. Cuando lanzamos una excepción, el **runtime** (la máquina virtual Java) va buscando en el call stack, en orden reverso de ejecución, si en alguno de los métodos hay un **exception handler** (o manejador de excepciones) asociado al tipo de excepción lanzado.

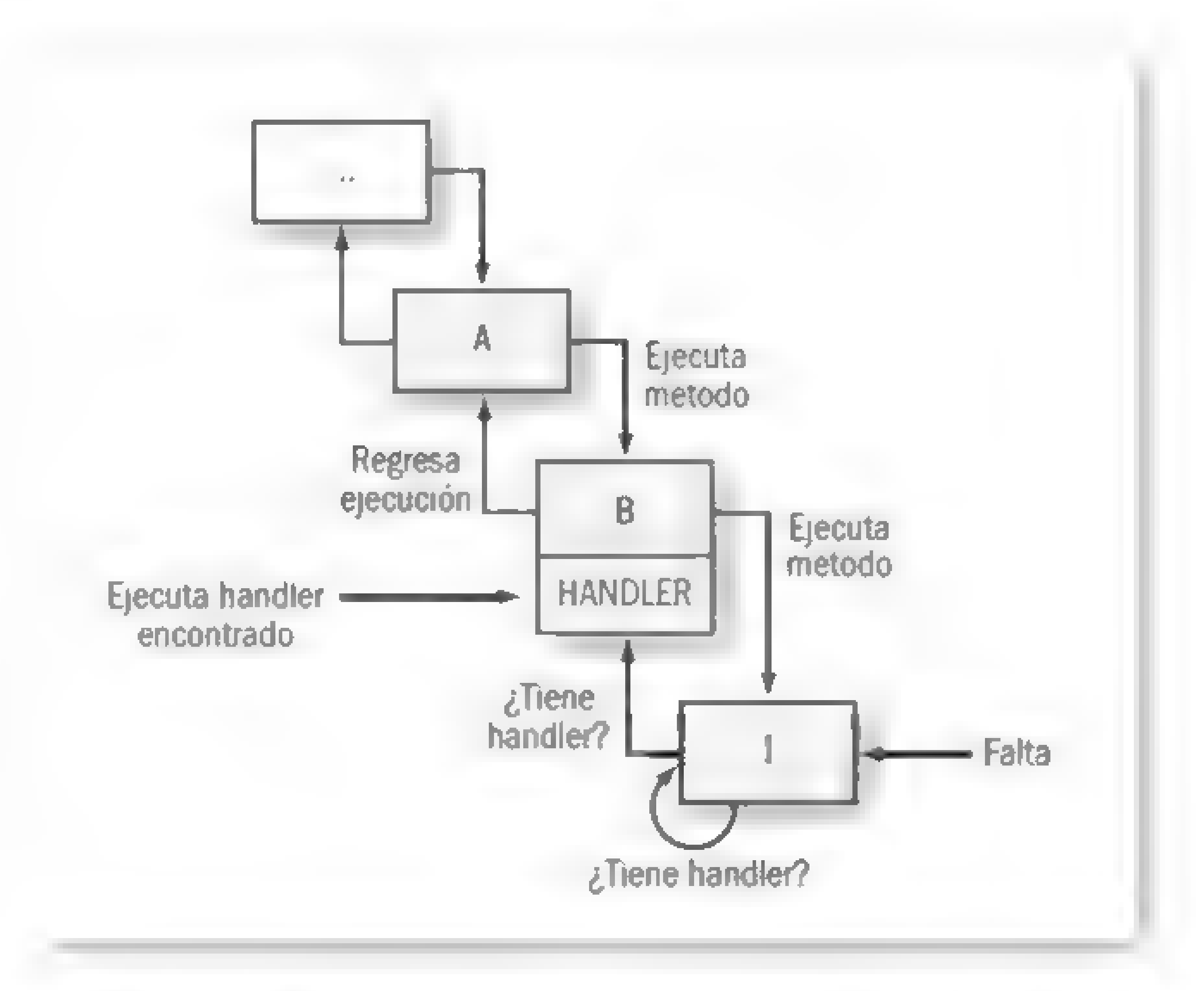


Figura 2. En un stack se busca la primera activación con un **handler** adecuado, yendo hacia atrás en la cadena

Cuando encuentra el primer handler, ejecuta su código asociado y regresa la ejecución al método al cual este pertenece, abortando la ejecución de los métodos del call stack que están entre el punto de error y el handler.

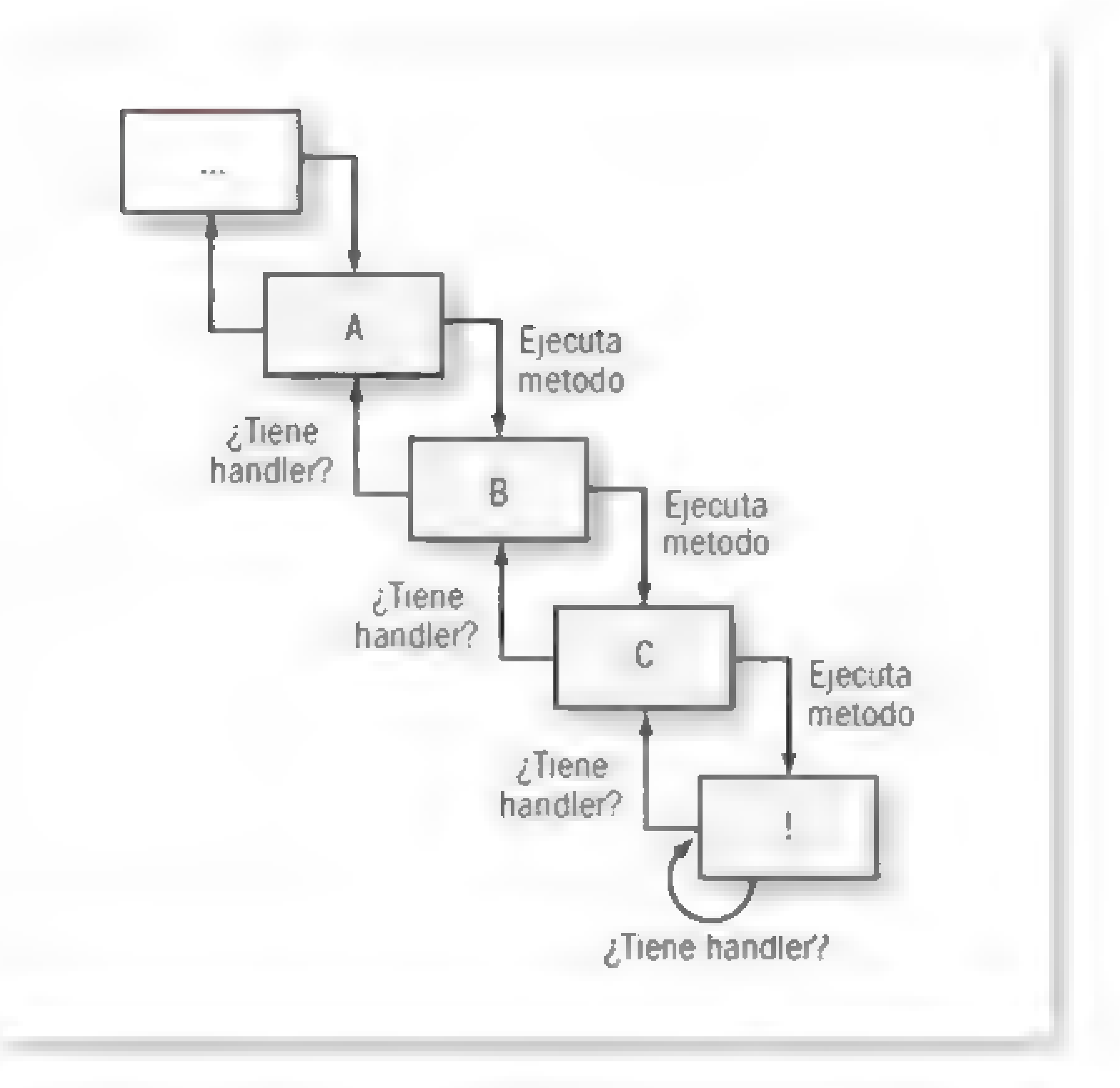


Figura 3. Cuando se busca un handler en el stack, si no se lo encuentra se sigue hasta llegar al handler por defecto, que termina la ejecución del programa.

Cuando el sistema no encuentra ningún handler apropiado, ejecuta su propio handler existente por defecto, que aborta la ejecución del programa. El hecho de encontrar un manejador apropiado y de ejecutarlo se conoce como **catcher** o **atrapar la excepción**.

En Java, los objetos que pueden ser lanzados como excepciones son del tipo Throwable o una de sus subclases. Esta clase tiene dos subclases que definen la estructura básica de todas las excepciones y errores. Por un lado, se encuentran los errores graves de sistema –que no deberían ser nunca de interés a una aplicación– representados por la clase Error y sus descendientes. Por otro, las situaciones excepcionales durante la ejecución de una aplicación, representadas por la clase Exception y

sus subclases. Es de particular interés la subclase RuntimeException, que generalmente representa errores en la lógica del programa que tienen un trato especial en el lenguaje.

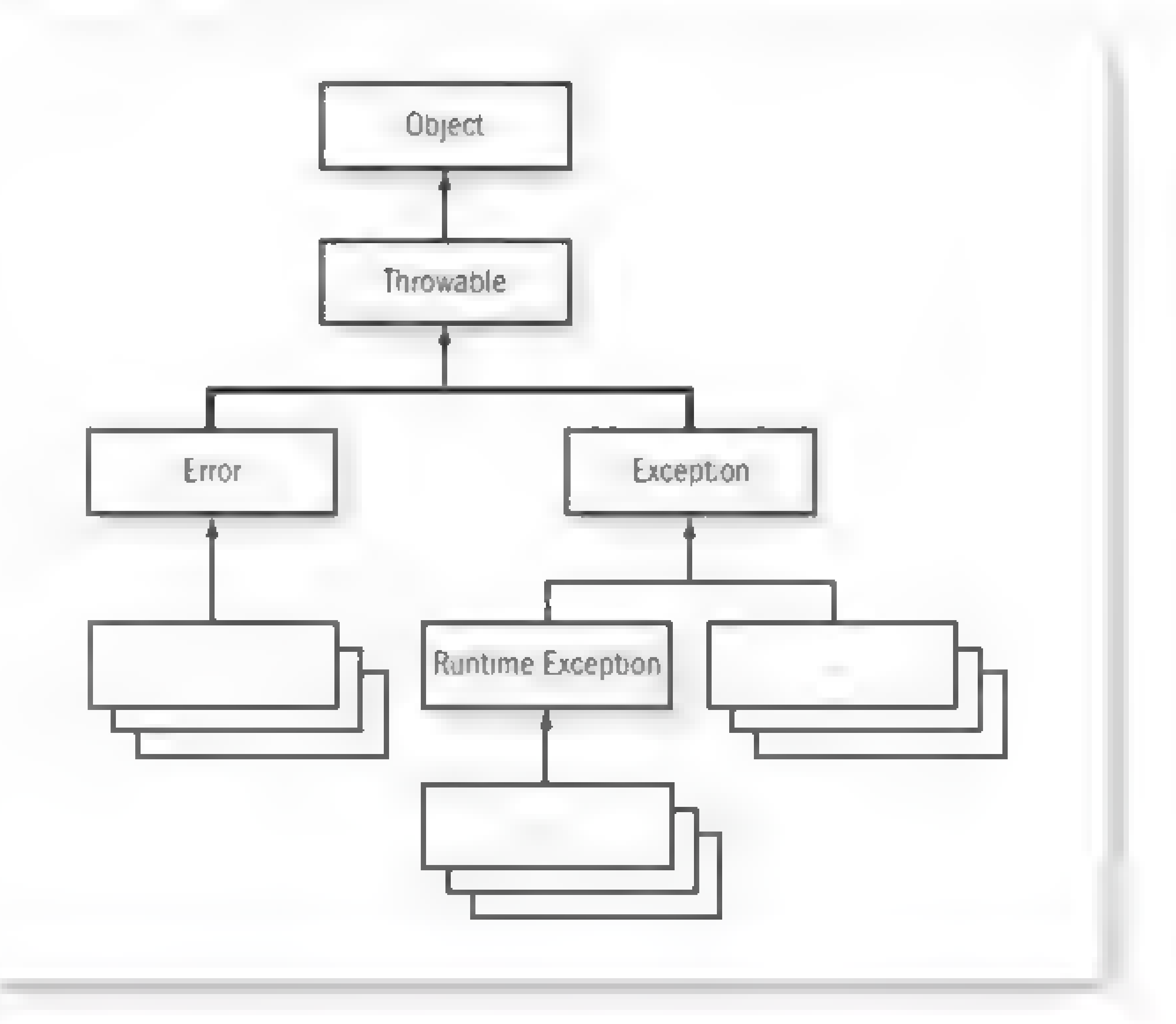
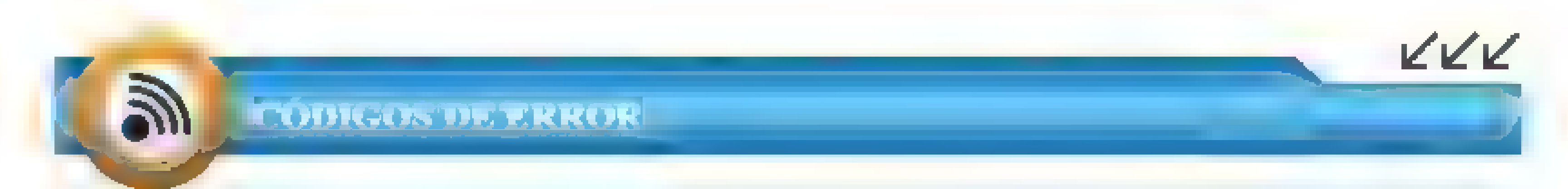


Figura 4. Jerarquía de las excepciones en Java. Throwable es la clase base para todas las demás. RuntimeException hereda de Exception.



Los lenguajes como C, sin excepciones, fuerzan al programador a verificar constantemente los resultados de las operaciones para saber si hubo un error o no. Los métodos regresan un dato que representa un código de error o una marca booleana para indicar éxito o fracaso. Así, el código sufre de gran cantidad de if y otros enredos para manejar los casos de error

Uso

Las excepciones son bastante sencillas de utilizar. Empecemos por ver cómo lanzamos una excepción:

```
throw new IllegalArgumentException("unParametro");
```

Utilizamos la palabra clave throw, seguida del objeto excepción que queremos lanzar. Generalmente, este es creado en ese mismo momento, como en el ejemplo. Cuando estamos interesados en manejar excepciones, tenemos que envolver el código que las puede provocar con la declaración try y catch.

```
try {
        ... // Código que en algún punto puede producir una excepción
} catch(TipoDeExcepcion excepcion) {
        ... // Código para manejar la situación de error
}
```

El código dentro del catch (handler) se ejecuta cuando el runtime decide que es el handler apropiado para tratar la excepción. Solamente pueden tratar excepciones que sean del tipo (o un subtipo) especificado, y dentro del bloque en que se tiene acceso al objeto excepción, como si fuera un parámetro.

Es posible definir varios handlers para varios tipos de excepciones en una misma declaración try.

```
try {
        ...
} catch(UnTipoDeExcepcion excepcion) {
        ...
} catch(OtroTipoDeExcepcion otraExcepcion) {
        ...
} catch(YOtroTipoMasDeExcepcion yOtraExcepcionMas) {
        ...
}
```

Se ejecutará solamente el primer handler asociado al tipo de excepción que ocurra. Si hay varios candidatos posibles (por ejemplo, si tenemos un handler para una subclase de una clase de excepción que también queremos manejar), entonces se ejecuta el primero definido. Por este motivo, los tipos más abstractos se definen por último.

Cabe aclarar que, dentro del bloque del manejador, solo se puede acceder a las variables declaradas fuera del try y al objeto excepción definido en el catch correspondiente.

Java provee una gran variedad de excepciones incluidas en el sistema, que es recomendable aprender y usar antes de crear nuevas excepciones. Algunas de las más utilizadas son las que se listan en la siguiente tabla:

EXCEPCIONES

EXCEPCIÓN	USO
IllegalArgumentException	Parametro no nulo invalido
IllegalStateException	El objeto receptor se encuentra en un estado invalido para ejecutar el método.
NullPointerException	Parametro nulo
IndexOutOfBoundException	Parametro para usar como indice en una coleccion que está fuera de rango.

Tabla 1. Excepciones más reutilizadas en general en los programas Java

Si queremos crear nuestras propias excepciones, lo recomendable es heredar de alguna de las clases ya existentes dentro de la jerarquía de Exception.

```
public UsuarioNoExistenteException extends Exception {
// El constructor recibe la información de lo ocurrido
        public UsuarioNoExistenteException(String nombre) {...}

// Hay métodos para obtener esa información
        public String nombreUsado() {...}
}
```

Es adecuado dotar a nuestra excepción de un constructor que acepte la información requerida para entender la causa del error y, también, proveer métodos para acceder a dicha información en el código de manejo.

Por otra parte, las excepciones pueden ser encadenadas en una relación causal. Para esto, podemos utilizar el constructor, para pasar la excepción que originó el error como parámetro, o utilizar el método initCause.

```
public void ingresarConUsuario(String nombre, String
 clave) throws IngresoNoPermitidoException {
        try {
                ...
        } catch(UsuarioNoEncontradoException e) {
                throw new IngresoNoPermitidoException(e);
        }
}
```

Este andamiento de excepciones es muy útil para abstraer al cliente de una API de errores internos. Abstraemos las excepciones y arrojamos una más significativa a la tarea que representa el método. También logramos disminuir el acoplamiento al reducir el número de excepciones que arroja un método (que forzamos a los métodos clientes a atrapar o arrojar) Es aconsejable que, cuando lanzamos excepciones, dejemos a los objetos involucrados en un estado utilizable. Con esto nos referimos a que, si manipulamos el estado de un objeto y el método falla, ese estado sea válido en la semántica del objeto, de modo que se le pueda seguir enviando mensajes y responda adecuadamente.

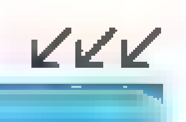
Excepciones chequeadas

Las excepciones que pertenecen a la jerarquía de Exception excepto las que son RuntimeException son conocidas como **excepciones chequeadas**. Son chequeadas porque el compilador Java fuerza a que se las trate específicamente. Cuando un método quiere arrojar una excepción de este tipo, debe declararlo en la firma del método. Para esto, se utiliza la palabra clave throws, seguida de las clases de las excepciones que se van a lanzar separadas por coma.

```
public Usuario buscarUsuarioCon(Nombre nombre)
throws UsuarioNoExistenteException {
        ...
        throw new UsuarioNoExistenteException(nombre);
        ...
}
public void guardar(Object algo) throws
ObjectoNoGuardableException, ErrorAlGuardarException {
        ...
}
```

El compilador nos alertará si no declaramos las excepciones en la firma del método. Otra opción es atrapar la excepción y tratarla.

Estas excepciones representan situaciones de errores esperables por la aplicación, y deberían ser recuperables. Por **recuperable** entendemos



En el mundo Java, es una convención que las excepciones tengan como sufijo la palabra Exception y que el nombre refleje adecuadamente la situación de error. También es normal facilitar varios constructores que aceptan información sobre la falla, un mensaje de error y, por último, la causa (excepción) del error.

que la aplicación puede tomar acciones para evitar el error en el futuro. Por ejemplo, si nuestra aplicación quiere leer un archivo, le pregunta al usuario por el archivo; y si resulta que este no existe, una de las opciones es preguntar de nuevo. O, si estamos en un sistema buscando un determinado usuario pero este no existe, el sistema podría preguntar si se desea crear el usuario. Generalmente estas excepciones representan errores en el dominio del problema y son tratables en la mayoría de los casos.

Excepciones no chequeadas

Las excepciones no chequeadas son del tipo RuntimeException o alguna de sus subclases. Son silenciosas: el compilador no nos obliga a atraparlas ni a declararlas en las firmas de los métodos (y, por lo tanto, no sabemos si un método arroja alguna excepción de este tipo).

Estas excepciones representan errores de la aplicación que son recuperables y, generalmente, resultan de errores de programación o de lógica en el programa. Un caso son las NullPointerException, que aparecen cuando queremos mandarle un mensaje a null.

Resultaría molesto que por todos lados nos obligaran a tratar o declarar estas excepciones, que podrían ocurrir en cualquier punto donde se envía un mensaje. Si bien es posible atrapar estos errores y tratarlos, debería ser en casos muy extraños y específicos. En la mayoría, hay que corregir el error en el programa.

Cuando estamos tratando con una excepción chequeada y no queremos arrojarla, debemos tratarla adecuadamente, es decir que debemos hacer algo con ella, y no simplemente ignorarla. Ignorar una excepción significa que su código manejador dentro de la estructura catch está vacío. Debemos evitar este tipo de código

Las excepciones chequeadas forman parte del contrato de un método, mientras que las no chequeadas representan lo que sucede si se lo rompe.

```
// No es necesario declararla ni atraparla
public void copiar(String unTexto) {
        if(unTexto == null) throw new IllegalArgumentException("unText");
        ...
this.copiar(algo);
// No sabemos que arroja una excepción
...
}
```

Manejo de errores

Los **errores** son problemas que ocurren por fuera de la aplicación, por lo que esta no puede anticiparlos ni tratarlos. Generalmente, se trata de problemas que encuentra el runtime con el sistema operativo. Por ejemplo, podemos estar leyendo un archivo y que se rompa el disco desde el cual lo estamos leyendo, o bien el sistema se quede sin memoria para crear nuevos objetos. Este tipo de errores no pueden ser manejados por la aplicación y terminan abortando el programa. Son objetos pertenecientes a la jerarquía de la clase Error y sus subclases.

Genéricos

En versiones de Java anteriores a la 5, las clases que tenían que estar preparadas para colaborar con objetos de distinto tipo usaban Object para referirse a ellos. Esto era una limitación, ya que obligaba a cotejar constantemente estos objetos aunque se supiera de qué tipo eran.

Para solventar esta situación, se agregaron los genéricos al lenguaje. Estos transforman las clases en una especie de plantilla donde, luego, el tipo correspondiente es inyectado, y de este modo evita los molestos cast.

En un esquema clásico de trabajo, se realizaban codificaciones como la siguiente (creación de una lista de strings):

```
List miLista = new ArrayList( );
miLista.add(“Linda”);
miLista.add(“Mañana”);
miLista.add(new Integer(2054));
```

Incluso se podía utilizar un casting al obtener los elementos de esa lista:

```
String s = (String) miLista.get(0);
String st = (String) miLista.get(1);
String str = (String) miLista.get(2);
```

Pero esto no evitaba que, al ejecutarse, se produjeran errores. Si leemos el código, el tercer elemento no es un String, sino un Integer, pero nada impide añadirlo a la lista. Es aquí donde los genéricos se presentan en el lenguaje para solucionar los inconvenientes de los desarrolladores y evitar errores de este tipo. Mediante el uso de genéricos, se logró que el compilador se asegure de que los tipos sean los correctos, y esto hizo desaparecer muchos errores en los programas. Para comprender mejor este tema, veamos otro ejemplo de cómo funcionan y para qué sirven los genéricos y analicemos qué problemas existían cuando no estaban en el lenguaje:

NO ABUSAR DE LAS EXCEPCIONES

Cuando agreguemos una excepción a la firma de un método pensemos si el cliente podrá hacer algo ante el error. Si no, arrojar la excepción expandirá la API y complicará al cliente.

```
Map menu = new HashMap();
menu.put("ravioles", new Double(14.56));
menu.put("carne al horno", new Double(32.00));
menu.put("sopa del dia", new Double(7.00));
...
// Al no estar tipado puedo poner cualquier cosa
menu.put(new Integer(1), new Auto());
// Tampoco sé qué hay adentro
(Float) menu.get("ravioles")
// Arroja un error al castear incorrectamente
```

Estos son algunos ejemplos de los problemas que generaba el hecho de que no se pudiera especificar el tipo de objetos que contenían las colecciones.

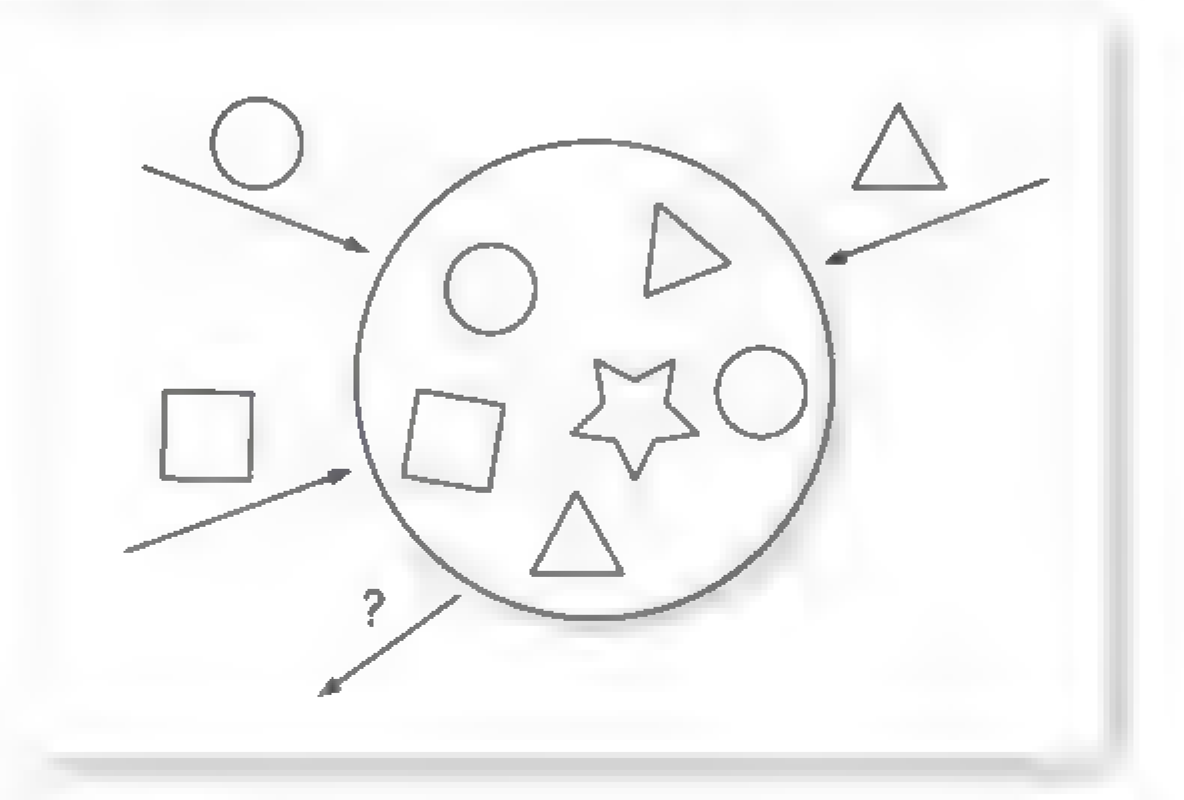


Figura 5. Así era el uso de las colecciones antes de la inclusión de genéricos en el lenguaje.

Otro ejemplo que encontramos en la librería básica de Java es la interfaz `Comparable`. Esta interfaz especifica la comparación entre objetos de un mismo tipo, de modo de determinar un orden natural entre ellos.

```
// Versión vieja
int compareTo(Object o)
// Versión nueva
int compareTo(T o)
```

Aquí vemos cómo el tipo `Object` fue reemplazado por el tipo `T`, un genérico. De esta forma, todo objeto que quiera implementar esta interfaz tiene que especificar a qué tipo corresponde `T`.

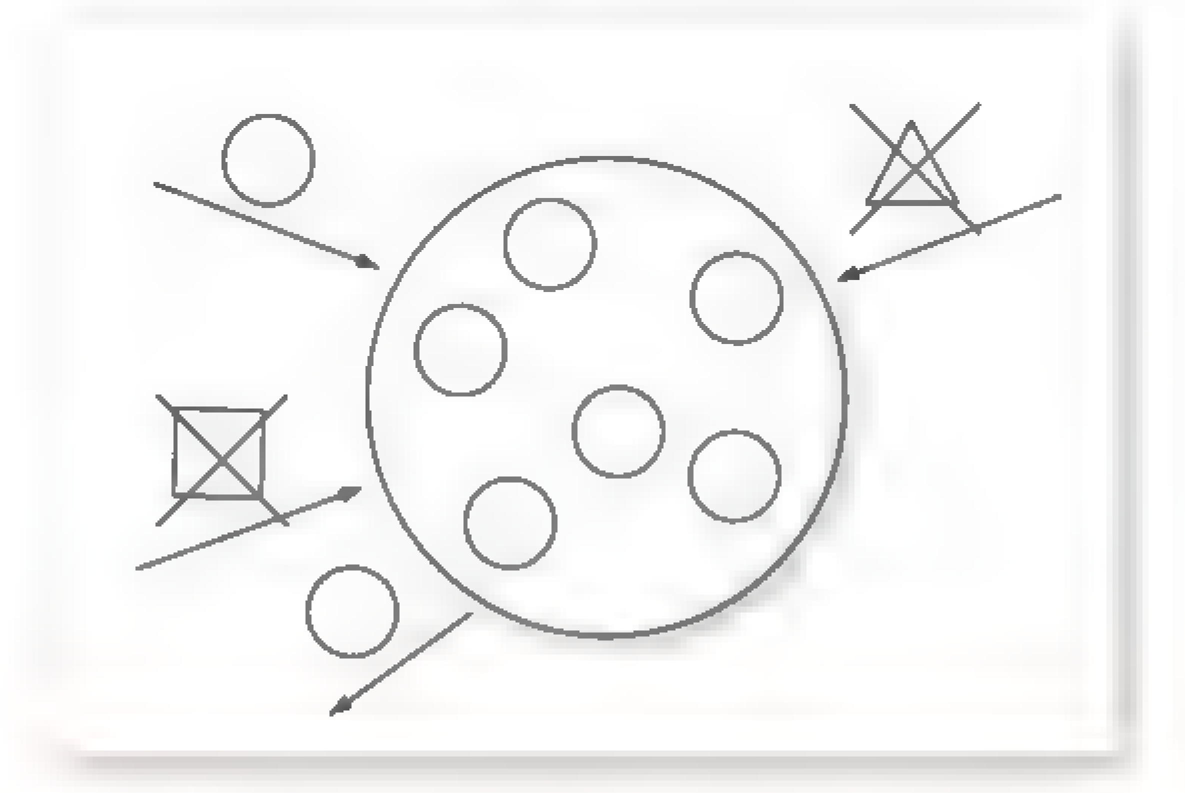


Figura 6. Los genéricos permiten tener colecciones genéricas especificadas para un tipo determinado.

Los genéricos existen en el código y durante la compilación, no así en el tiempo de ejecución, pues el compilador borra toda la información relacionada. Así, no hay problemas con aplicaciones antiguas y el código resultante es fiable porque el compilador lo verifica

Se logra que el compilador verifique que los objetos por comparar tengan el tipo adecuado. También se evita tener que chequear que el objeto pasado como parámetro sea instancia del tipo que interesa, reduciendo así el código y los errores.

Para definir un tipo genérico debemos enumerar, luego del nombre de clase o interfaz, cuántos tipos genéricos vamos a tener y cómo vamos a referenciarlos en el resto del código. Los tipos genéricos se declaran separados por comas entre los símbolos <y >.

```
public interface Comparable<T> {
...
}
```

Luego, en el código podemos utilizar el tipo T como cualquier otro tipo:

```
public interface Comparable<T> {
        int compareTo(T o);
}
```

Generalmente, para denominar a los tipos genéricos, se usa una sola letra en mayúscula y se utilizan las últimas letras del abecedario, comenzando por la letra T.

EXCEPCIONES EN JAVAWORLD

Si nos dirigimos, en nuestro navegador favorito, a **www.javaworld.com** y buscamos artículos sobre excepciones, encontraremos que existe una gran cantidad de ellos. Es recomendable leer unos cuantos para interiorizarnos sobre este tema y sobre el funcionamiento del **stack**.

También se suele utilizar, cuando hay varios genéricos involucrados, la primera letra del concepto que representan. Por ejemplo, K para key (clave) y V para value (valor).

Cuando queremos utilizar un tipo genérico (instanciar) tenemos que especificar con qué tipos lo vamos a hacer.

```
Map<Producto,Precio> catalogo = new LinkedHashMap<Producto,Precio>();
catalogo.put(jabon, new Precio(2.50));
...
Precio precio = catalogo.get(jabon);
...
// Error de compilación
Double precio = catalogo.get(acondicionador);
...
// Error de compilación
catalogo.put("body shower", new Precio(2.35));
```

No solo podemos especificar tipos genéricos en la definición de un tipo. También podemos utilizar un tipo genérico específicamente para un método. Para ello, declararemos el o los tipo/s genérico/s como modificador es del método. Esto es válido para el alcance del método, su código, sus parámetros y su tipo de retorno.

Puede pasar que trabajemos con código viejo y que use los tipos sin especificar el genérico En estos casos, el compilador nos lo hara notar mediante un alerta. Podemos informar al compilador que estamos seguros de lo que hacemos y evitar el alerta mediante la anotacion @SuppressWarnings con los parametros unchecked o rawtypes.

```
public class Buscador<T> {
...
        public <U> T[] filtrar(U base) {
                ...
        }
...
}
```

En el ejemplo, el tipo U es **inferido** por el compilador y depende del contexto de uso; si pasamos un String, entonces U será String.

Este uso es muy conveniente para poder forzar el mismo tipo en varios parámetros y, al mismo tiempo, en el tipo de retorno.

Subtipado

Una característica que en la primera impresión resulta extraña es el **subtipado** de genéricos. Por ejemplo, si tenemos Habitat<Felino> y Habitat<Tigre>, no existe relación alguna de subtipado entre estos dos.

```
// Inválido
Habitat<Felino> habitat = new Habitat<Tigre>();
```

Las excepciones chequeadas son aquellas que heredan de Exception, excepto las que lo hacen de RuntimeException. Si deseamos implementar nuestras propias excepciones no chequeadas y no deseamos heredar de RuntimeException, podemos extender Throwable. Pero no solamente podemos crear errores, sino también alguna forma de control de flujo ajena a las normales. No es recomendable.

Pensemos por qué no tiene sentido que exista un relación. Si el ejemplo anterior fuese correcto, podríamos agregar un puma a un hábitat de tigres. El principio de sustitución, que es la base del subtipado, se ve comprometido.

Comodín

En el uso de los genéricos ocurre, en algunas ocasiones, que no nos interesa el tipo genérico o el tipo específico. Podemos entonces especificar esto con un **comodín** en lugar del tipo; para ello, usaremos el símbolo ?. Este símbolo indica cualquier tipo (aunque no significa que sea igual a Object).

```
public Cuidador {
        void llevarAlimentoA(Habitat<?> habitat) {
        ...
}
}
```

Tipos restringidos

En ciertas ocasiones, queremos que los tipos permitidos para un genérico pertenezcan a alguna jerarquía, con el fin de poder usar ciertos métodos propios a ella en el código genérico.

Si nuestros metodos comparten mucho código de manejo de excepciones similares, lo más conveniente es crear un metodo **template** Este metodo contendra el código común para manejar excepciones y delegara la funcionalidad diferente a otros metodos Estos metodos pueden estar en la misma clase o pertenecer a otra, segun lo requiera el problema

Por ejemplo, si tenemos la clase que representa hábitats en un zoológico y lo hacemos genérico, esperamos que lo que pongamos allí sea un animal y no un número. Por este motivo, especificamos que los tipos permitidos tienen que ser Animal o uno de sus subtipos. Lo hacemos utilizando la palabra extends y decimos que estamos definiendo un **techo** para los tipos permitidos.

```
public class Habitat<A extends Animal> {
 ...
}
```

En el código de esta clase, todo objeto que sea de tipo A será tratado como un Animal, pudiendo enviarle los mensajes que entiende un tipo Animal.

Por otra parte, si queremos que dicho tipo genérico extienda de varios tipos simultáneamente (que implemente una o varias interfaces que nos interesan), podemos especificarlo al conectar cada tipo que se quiere extender con el símbolo &.

```
public class CajonDeVerduleria<V extends Verdura & Fruta> {
 ...
}
```

DEFINIR UN PISO PARA LOS TIPOS

Existe también una forma de restringir los tipos utilizando la palabra super. Esta permite especificar que aceptamos cualquier tipo que sea un supertipo del especificado (incluido él mismo). Esta restricción es utilizada en casos muy particulares y su concepto es difícil de entender.

También podemos utilizar el comodín junto con el extends. Si tenemos un método para transportar animales de un hábitat a otro, podemos utilizar el comodín con el extends para indicar que esperamos un hábitat de algún subtipo de Animal, inclusive Animal. Si pusiéramos directamente un hábitat de Animal, solamente podríamos pasar este tipo de hábitat y no, por ejemplo, un hábitat de tigres.

```
public class Habitat<A extends Animal> {
        void transferirA(Habitat<? extends A> habitat) {
                ...
        }
}
```

Genéricos en el alcance estático

Un uso interesante de los genéricos se da cuando son utilizados por métodos estáticos. Veamos un ejemplo concreto:

```
public class Colecciones {

        public static <E> List<E> unaListaCon(E ... elementos) {
                ...
        }

}
```

Tenemos un **método genérico estático** que no tiene ninguna particularidad, salvo cuando lo usamos:

```
import static Colecciones.*;
…
…
List<String> nombres    unaListaCon("Pedro", "Juan", "Lucia");
```

Gracias a los genéricos, tenemos un método para poder crear listas enumerando sus elementos. El compilador se encarga de realizar todos los chequeos de tipado para que el uso sea correcto.

Este tipo de métodos estáticos son muy útiles para la construcción de objetos complejos, como en el ejemplo. Escribir uno mismo el código para tal cosa es tedioso, largo, requiere especificar tipos y puede tener errores.

Las excepciones son un poderoso mecanismo para indicar fallas en la ejecución de un programa, ya que permiten escribir un bloque de colaboraciones e indicar luego qué se debe hacer ante un determinado error. Por otra parte, los genéricos fueron un gran aporte al lenguaje Java, ya que permitieron mejorar notablemente el código y eliminaron errores debidos a los casteos y al mal uso de las colecciones. Los genéricos permiten escribir código más seguro y claro, al evidenciar las restricciones de tipo que el programador tiene en su mente.

Actividades

TEST DE AUTOEVALUACIÓN

1 ¿Qué es una **excepción**?

2 ¿Cómo se lanza una excepción?

3 ¿Qué ocurre cuando una excepción es lanzada?

4 ¿A qué llamamos **call stack**?

5 ¿De qué forma se atrapa una excepción?

EJERCICIOS PRÁCTICOS

1 Haga una prueba para ver qué información se obtiene en un **stack trace**. Vea el método getStackTrace de Throwable.

2 Pruebe cambiar las excepciones usadas en el proyecto de ejemplo "El Juego de la Vida" (Game of Life) por algunas propias que sean chequeadas. ¿Cuánto más código tiene que agregar para manejarlas?

3 Busque en la librería Java alguna **excepción abstracta**. ¿Tiene sentido tal cosa?

Si tiene alguna consulta técnica relacionada con el contenido, puede contactarse con nuestros expertos: **profesor@redusers.com**.

Librería base

Conocer las herramientas disponibles para realizar un trabajo es fundamental, ya que permite determinar cuál se adecua mejor a la tarea. Por este motivo, quien desea aprender Java debe conocer las clases e interfaces que se proveen desde la instalación. No solo se ahorra trabajo al utilizar algo ya existente, sino que el código puede comunicar mejor su propósito a otros programadores, ya que usa elementos estándares.

↗↗↗

Librería y objetos básicos

Java posee una amplia librería base que está disponible con la instalación inicial, ya sea del kit de desarrollo o del entorno de ejecución. Esta librería provee lo necesario para desarrollar todo tipo de aplicaciones, desde las de escritorio hasta aplicaciones web. Permite crear ventanas, escribir archivos, conectarse a un servidor y muchas cosas más.

Comenzaremos nuestra exploración de la librería por las clases que conforman los ladrillos básicos sobre los cuales se construyen las demás clases del sistema.

java.lang.Object

Esta clase, como ya sabemos, es la clase padre de Java, de la cual todas las demás heredan. Aquí encontramos definidos los métodos más primitivos y generales que podemos esperar de los objetos. Lo más básico que podemos saber de un objeto en Java es lo siguiente: si es igual a otro objeto (equals), a qué clase pertenece (getClass) y acceder a una representación en texto (toString).

```
Class<?> getClass()
boolean equals(Object otroObjeto)
String toString()
```

Java ofrece la clase java.util.Arrays, que brinda métodos estáticos para ordenar, buscar, comparar, copiar y crear arrays fácilmente. Estos tienen versiones sobrecargadas para los arrays de tipos primitivos y versiones genéricas para los objetos.

Otro método muy importante, pero que a primera vista no nos transmite su utilidad, es hashCode. Es un número entero (int) usado por ciertas colecciones, generalmente los mapas (diccionarios), para guardar el objeto de modo que el hashCode sea índice de este.

```
int hashCode()
```

Finalmente, Object provee otros métodos que no son muy utilizados, al menos en la gran mayoría de las situaciones. Uno de ellos es el método para clonar (duplicar) un objeto, no tan utilizado porque la clase de los objetos por clonar debe implementar la interfaz Cloneable y tener ciertas consideraciones. Los demás métodos definidos son para sincronizar, de forma rudimentaria, varios hilos de ejecución, pero es recomendable usar otras maneras de sincronización más avanzadas.

Implementar los métodos equals y hashCode

Los métodos equals y hashCode son especiales y requieren cuidado a la hora de ser sobrescritos en las otras clases. El método equals requiere que se implemente una relación de **equivalencia**: reflexiva, simétrica y transitiva. Es **reflexiva** si para todo objeto no nulo x, x.equals(x) retorna true. Es **simétrica** si se cumple que para todo par de objetos no nulos x e y, x.equals(y) es igual y.equals(x). Finalmente, es **transitiva** si, y solo si, para los objetos no nulos x, y y z se cumple que si x.equals(y) regresa true a su vez que y.equals(x) también lo hace, entonces x.equals(z) tiene que regresar true. Se espera que el método sea **consistente**; esto quiere decir que sucesivas invocaciones del método con los mismos parámetros arrojen los mismos resultados (a menos que el objeto cambie). Finalmente, todo objeto x no nulo es distinto a null y por lo tanto x.equals(null) da false.

El método hashCode utiliza la información del objeto (sus atributos) y genera un número entero utilizado en las colecciones que usan **tablas de hash** internamente. La idea es que, utilizando los atributos que definen la identidad

del objeto (generalmente, los mismos que se utilizan en el equals), se calcule un número que sea bien distinto al generado por otro objeto ligeramente parecido. Objetos iguales deben devolver el mismo número, pero objetos que devuelven el mismo número no tienen que ser necesariamente iguales (siempre habrá objetos distintos con el mismo número cuando hay más de 4294967295 objetos). El cálculo debe ser rápido mientras el objeto no cambie.

java.lang.Boolean

La clase Boolean modela los valores de verdad y falsedad booleanos y en sus instancias encapsula los valores primitivos boolean, true y false. Esta clase provee métodos estáticos para pasar de String a boolean y viceversa.

```
static String toString(boolean b)
static Boolean valueOf(boolean b)
static Boolean valueOf(String s)
static boolean parseBoolean(String s)
```

java.lang.Number, hijos y java.lang.Math

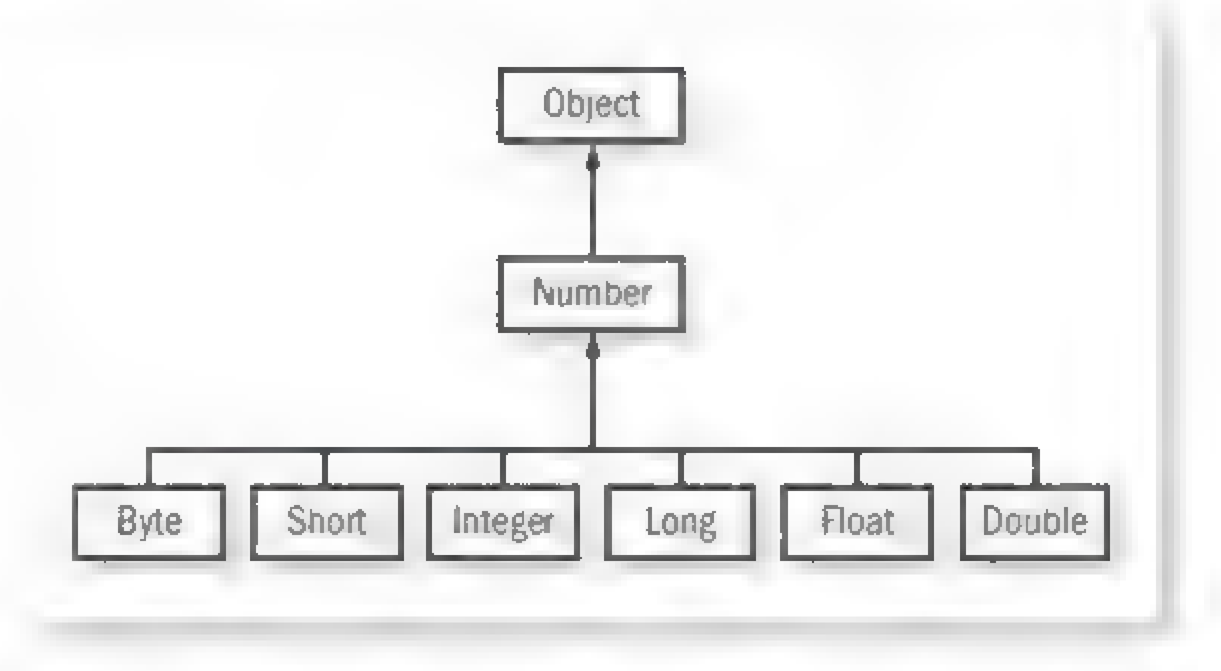


Figura 1. Jerarquía de Number y los distintos tipos de números

Number es la superclase de todos los tipos de números que hay en Java. Esta superclase solamente define métodos que permiten obtener, a partir de un objeto numérico, los distintos valores primitivos.

```
byte byteValue()
short shortValue()
int intValue()
long longValue()
float floatValue()
double doubleValue()
```

Sus subclases más conocidas son aquellas asociadas a los tipos numéricos primitivos: Integer, Byte, Short, Float, Long y Double. Estas clases proveen métodos para manipular los números, además de varias formas de obtener el número a partir de un String.

Por su parte, la clase Math es un compendio de los métodos abstractos que implementan la mayoría de las funciones matemáticas que encontramos en cualquier calculadora científica. Primero, posee las constantes matemáticas **Pi** y **E** y, luego, ofrece métodos para las operaciones trigonométricas, para hacer cálculos exponenciales, redondeos y obtener números aleatorios, entre otros.

```
static double E = 2.7182818284590045d;
static double PI = 3.141592653589793d;
static int min(int a, int b)
static double sin(double radianes)
static double toRadians(double grados)
static double random()
static double log(double a)
static double exp(double a)
```

java.lang.String y java.lang.Character

Character es la clase asociada con los char y define los caracteres como objetos. Java soporta los caracteres Unicode y todas sus representaciones. La clase provee varios métodos para caracteres a partir de algún código Unicode, varios métodos para categorizar al carácter y para pasar de minúscula a mayúscula y viceversa.

Los String representan cadenas de caracteres y tienen su forma literal tipo **abc**. Son inmutables, o sea que sus valores (los caracteres) no pueden cambiar con el tiempo; cualquier método que opere sobre ellos devuelve un nuevo String. Esta clase tiene la mayoría de los métodos que necesitamos para manejar cadenas de caracteres; algunos que mencionaremos a continuación son para saber su longitud, conseguir una porción de él, buscar pedazos de texto dentro de la cadena, concatenar más texto y reemplazar secciones.

```
char charAt(int indice)
String concat(String texto)
boolean equalsIgnoreCase(String texto)
boolean isEmpty()
int length()
replace(String expresionRegular, String reemplazo)
```

EXPRESIONES REGULARES

Las expresiones regulares, también conocidas como **regexs**, son una notación para definir pequeños lenguajes o patrones. Generalmente se escriben como un String y son muy versátiles para realizar búsquedas y verificaciones de forma en textos Java ofrece un muy buen soporte para **regex**. Si vemos la documentación asociada a la clase java.util.Pattern, tendremos una breve introducción a este tema.

Hay dos clases asociadas a los strings: StringBuilder y StringBuffer, que se utilizan cuando se quiere construir o modificar strings de gran tamaño de forma eficiente en cuanto a memoria y velocidad. Las operaciones principales de estas clases son Insertar (insert) y Agregar (append). Veremos estas clases en detalle más adelante, en este mismo capítulo.

Colecciones

Las **colecciones** son elementos fundamentales de cualquier programa, ya que permiten manejar agrupaciones de objetos de forma simple y consistente.

Existen distintos tipos de colecciones que modelan de forma distinta el agrupamiento de los objetos y tienen propósitos bien distintos. Es conveniente conocerlas bien para poder elegir la mejor para cada tarea. Comencemos por las clases e interfaces más simples.

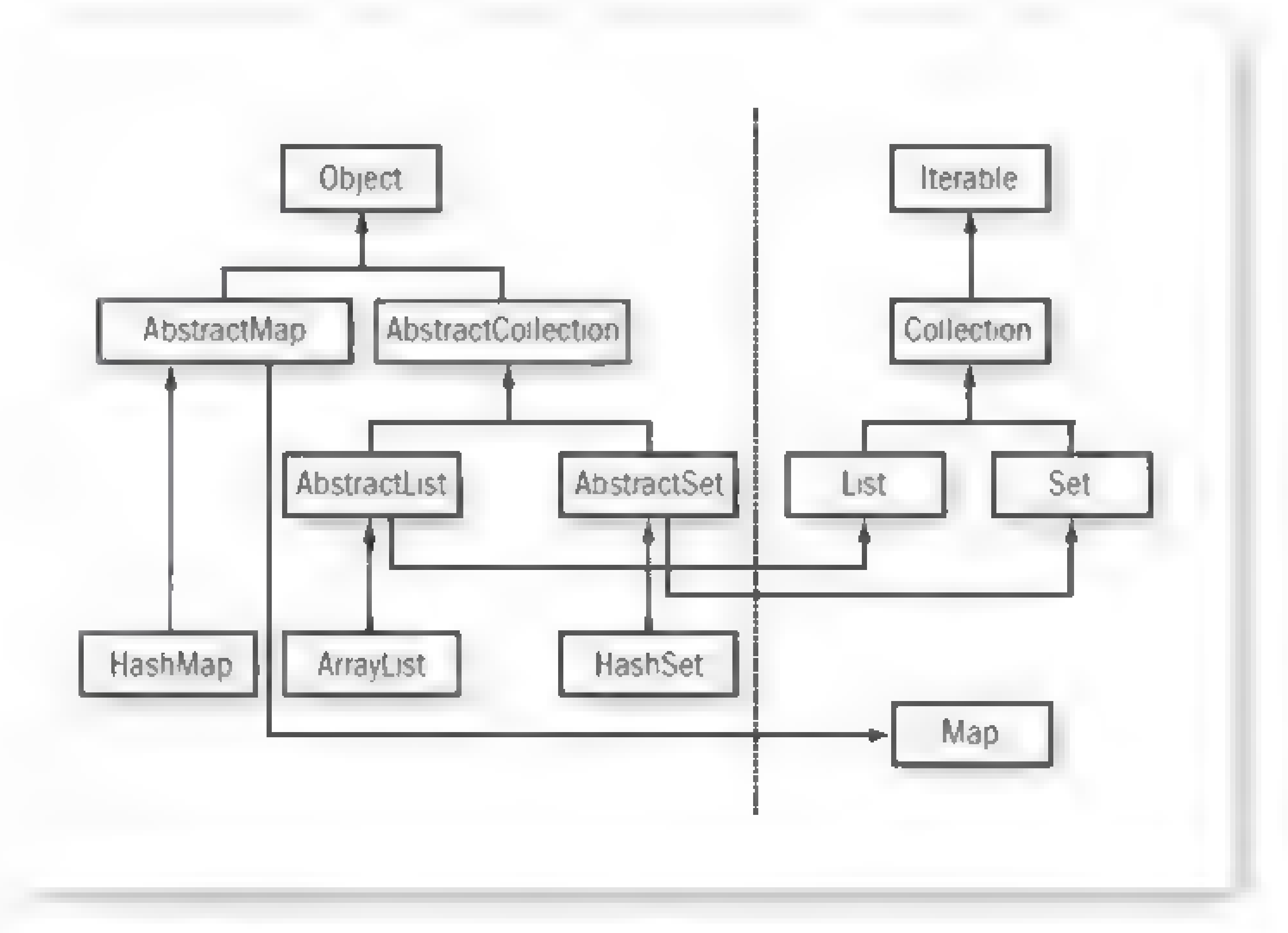


Figura 2. Jerarquía de las distintas colecciones de Java.

java.util.Iterable y java.util.Iterator

Estas interfaces permiten que las agrupaciones de objetos (por ejemplo, las colecciones) sean recorridas de a un elemento por vez. Todas las colecciones implementan Iterable, que permite que sean utilizables en la estructura for each.

Todo Iterable da acceso a un Iterator, que es un objeto encargado de realizar el recorrido de los objetos. También permite remover un elemento mientras se realiza el recorrido sin incurrir en un error, pero no permite hacer inserciones o agregar objetos nuevos mientras se recorre.

Veamos el protocolo ofrecido por Iterable:

```
Iterator<T> iterator()
```

Y a continuación, el de Iterator:

```
boolean hasNext()
T next()
void remove()
```

Siempre hay que llamar a hasNext() antes que a next() para asegurarse de que hay más elementos para ver y no obtener un excepción.

Se preve el agregado de un nuevo uso al try para que permita operar con streams sin tener que cerrarlos correctamente. Esta forma se conoce como **try with resource** y es, sencillamente, una manera mas facil de escribir el mismo código (try y luego un finally para cerrar el stream)

java.util.Collection

Es la interfaz raíz de todas las colecciones en Java (excepto de los diccionarios o mapas) y modela las agrupaciones de objetos sin especificar si estos tienen un orden o no, o si se acepta el mismo objeto varias veces o, incluso, si soporta null entre sus elementos. Lo que sí define es un protocolo mutable para colecciones, pero deja claro que los métodos para realizar los cambios son opcionales (arrojando UnsupportedOperationException)

Este tipo de interfaz no es un buen diseño, ya que puede llevar a obtener un error en la ejecución. Resulta mejor separar esta interfaz en dos o más: una para los protocolos inmutables y otra para los mutables. De esta forma, se puede definir mejor qué tipo de colección se ofrece y qué tipo se requiere.

Los métodos ofrecidos permiten agregar elementos, removerlos, saber si están o no en la colección, conocer la cantidad de elementos presentes y conseguir un iterador para recorrerlos.

```
boolean add(T elemento)
// Devuelve true si se agregó
void clear()
boolean contains(Object o)
boolean isEmpty()
Iterator<T> iterator()
boolean remove(T elemento)
// Devuelve true si se remueve
int size()
Object[] toArray()
```

En **www.functionaljava.org** hallaremos conceptos del paradigma funcional a Java. Algunos de los objetos involucrados modelan funciones y colecciones con metodos para mapear y filtrar, entre otras operaciones. Las implementaciones son inmutables y configurables

java.util.List

Las **listas** son colecciones que disponen sus elementos por orden de llegada; colecciones ordenadas, también conocidas como **secuencias**. Los elementos tienen un **índice numérico**, como los array, y es posible agregar elementos en cualquier posición. El acceso a ellos se realiza mediante dicho índice, donde el primer elemento tiene índice 0. Generalmente, las listas soportan que estén duplicados y realizan búsquedas sobre sus elementos.

List ofrece un **iterador** distinto (además del común a todas las colecciones), que permite no solo remover sino también agregar y reemplazar elementos, además de ofrecer navegabilidad hacia atrás.

```
void add(int indice, T elemento)
T get(int indice)
int indexOf(Object o)
int lastIndexOf(Object o)
ListIterator<T> listIterator()
T remove(int indice)
// Devuelve el elemento removido
T set(int indice, T nuevo)
// Devuelve el viejo elemento
List<T> subList(int desde, int hasta)
```

MODELAR CON COLECCIONES

Al diseñar, tenemos que lograr un isomorfismo entre el dominio y lo que programamos Por este motivo, las colecciones no se tienen que utilizar para modelar conceptos, sino como implementación soporte de tal modelo. Por ejemplo, el menú de un restaurant no es simplemente una asociación de un plato con un precio Diseñar así lleva a poner, equivocadamente, una lógica de manipulacion en otros objetos.

java.util.Set

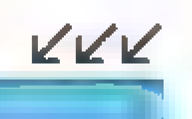
Set representa un conjunto de elementos. Esto quiere decir que es una colección que no acepta duplicados y que no define un orden para los elementos que contiene. Esta interfaz no define ningún protocolo extra al de Collection, pero especifica la semántica de aquellos (usando documentación, lamentablemente).

Las implementaciones de Set se aseguran que solamente se agregue una vez un objeto determinado –generalmente, usando equals y hashCode– pero solo en el momento de agregarlo. Si un objeto que pertenece a un Set muta (cambia sus atributos), no se asegura que se mantenga la invariante de que dicho Set no contenga dos elementos iguales. Por lo tanto, hay que asegurarse de utilizar objetos inmutables o de no cambiarlos mientras se los utiliza en este tipo de colecciones.

java.util.Map

Esta interfaz representa un conjunto de asociaciones entre pares de objetos, donde uno de ellos es considerado como clave. Por lo común, se denomina a este tipo de colecciones como **diccionarios** o **mapas** y, extrañamente, en Java esta interfaz no hereda de Collection.

Los objetos usados como claves conforman un conjunto y por lo tanto no pueden usarse para asociar dos objetos distintos al mismo tiempo en el mismo mapa (y, al igual que con los Set, deberían ser inmutables). Map tampoco extiende Iterable, por lo tanto los objetos de este tipo no pueden ser usados



APACHE COMMONS

En el sitio **http://commons.apache.org** encontraremos una gran cantidad de librerías, que dan funcionalidades para colecciones y para tratar con los tipos primitivos. También se ofrecen validadores, parsers, clases auxiliares para I/O y mucho más

en el for each. En cambio, Map ofrece tres modos de acceder a su contenido, como si fuera una colección. Primero, podemos acceder a las asociaciones mediante el mensaje entrySet, que devuelve un Set<Map.Entry<K,V>> (o sea, un conjunto de pares clave-valor). Segundo, para acceder al conjunto de claves, podemos enviarle al mapa el mensaje keySet. Finalmente, si solamente nos interesan los valores y no las claves, podemos utilizar el método values, que retorna una colección con los valores.

Para agregar elementos a un mapa se utilizan los métodos put y putAll, donde el primero espera como parámetros la clave y el valor, mientras que el segundo espera otro mapa. Para remover elementos se usa el método remove, que recibe la clave. Este método remueve la asociación completa, clave y valor.

Si queremos obtener el valor asociado a una determinada clave, usamos get, que toma como argumento la clave y devuelve el valor asociado. Si no está asociada la clave a ningún valor en el mapa, se retorna null. Podemos consultar si existe una clave en particular mediante containsKey, y un valor mediante containsValue. Extrañamente, el método remove acepta cualquier objeto como clave para remover la asociación, pero si no es del tipo de clave apropiado arroja una excepción. Sería más correcto evitar este error forzando la firma del método a que la clave fuera del tipo correcto.

```
V put(K clave,V valor)
  Regresa el valor asociado anterior o null si no había
void putAll(Map<K,V> mapa)
void clear()
boolean isEmpty()
int size()
Collection<V> values()
Set<Map.Entry<K,V>> entrySet()
Set<K> keySet()
boolean containsKey(K clave)
boolean containsValue(V value)
V get(K clave)
V remove(Object o)
// Devuelve el valor removido o null si no existe tal asociación
```

La interfaz Entry está definida en el contexto de Map, ya que está íntimamente relacionada a ella. Este es un uso claro para definir interfaces (o clases) anidadas (y estáticas, ya que es pública).

Ejercicio: colecciones diferentes

En este ejercicio, queremos obtener unas colecciones que permitan recorrerlas, filtrarlas y manipular sus elementos fácilmente. Las colecciones de Java delegan el trabajo al cliente, lo que hace que se escriba mucho código repetitivo y propenso a errores. Lo que buscamos es que las propias colecciones se encarguen del código repetitivo y que sean configurables (por ejemplo, que el ordenamiento y la semántica de la igualdad sean parametrizables).

Veamos el siguiente ejemplo con List y Collection. Crearemos un JUnitTest, lo nombraremos TestColeccion y luego crearemos las diferentes colecciones para testear con el siguiente código:

```
// Importamos las librerias
import java.util.*;
import java.util.List;
import java.util.ArrayList;
import java.util.Collection;

public class TestColeccion {

        @Test
        public void test() {

                // LIST
                int arr[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
                List<int[]> lista = Arrays.asList(arr);
                System.out.println(lista.size());

          assertNotNull(lista);

                List<Integer> numeros   Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
```

```
            assertNotNull(numeros);

            List<String> numerosTexto = Arrays.asList("uno","dos","tres");
            assertNotNull(numerosTexto);

            // COLLECTION
    Collection<String> listaMarcasCoches = new
ArrayList<String>(); // El tipo de listaMarcasCoches es Collection
    listaMarcasCoches.add("Audi");
    listaMarcasCoches.add("Porsche");
    listaMarcasCoches.add("Aston Martin");
    listaMarcasCoches.add("Ferrari");
    listaMarcasCoches.add("Mercedes");
    listaMarcasCoches.add("Seat");
    System.out.println("Número elementos
antes de eliminar: " + listaMarcasCoches.size() ) ;
    System.out.println (listaMarcasCoches.toString() ) ;
    listaMarcasCoches.remove ("Seat");
    listaMarcasCoches.remove ("Mercedes");
    System.out.println("Número elementos después
de eliminar Seat y Mercedes:" + listaMarcasCoches.size() ) ;
    System.out.println(listaMarcasCoches.toString () );


     };
}
```

La clase java.util.Collections es un repositorio de métodos estáticos que permiten realizar operaciones sobre las colecciones que no están disponibles. Por ejemplo, ofrece metodos para buscar los elementos maximos y minimos y ordenarlos. Tambien permite ampliar las colecciones y hacerlas inmutables o sincronizables.

Clases útiles

Existen muchas clases que pueden sernos útiles para diferentes propósitos. En esta sección conoceremos algunos ejemplos.

java.util.Date y java.util.Calendar

Java combina la noción de fecha y hora en una sola clase llamada Date. Es necesario utilizar un calendario, clase Calendar, para poder crear y manipular instancias de Date.

Entre algunos de los problemas que tiene Date están la mutabilidad y la no internacionalización. Igualmente, esta clase es ampliamente utilizada en librerías y productos, ya que forma parte de la librería base. Veamos algunos ejemplos del uso de Date con Calendar:

```
// Para conseguir la fecha actual
Date ahora = new Date();
// O si no
Date ahora = Calendar.getInstance().getTime();
```

La recomendación es evitar utilizar estas clases, ya que no están bien diseñadas y su uso es difícil y engorroso. Conviene utilizar alguna librería específica para manejar fechas y horas, como Joda Time y Date4J.

En **http://code.google.com/p/guava-libraries** se encuentra el proyecto de **Google** similar a **Apache Commons**. Este proyecto incluye una enorme cantidad de implementaciones de colecciones eficientes para distintos tipos de uso. Asimismo, contiene métodos auxiliares para trabajar con los tipos primitivos, con archivos y con la red.

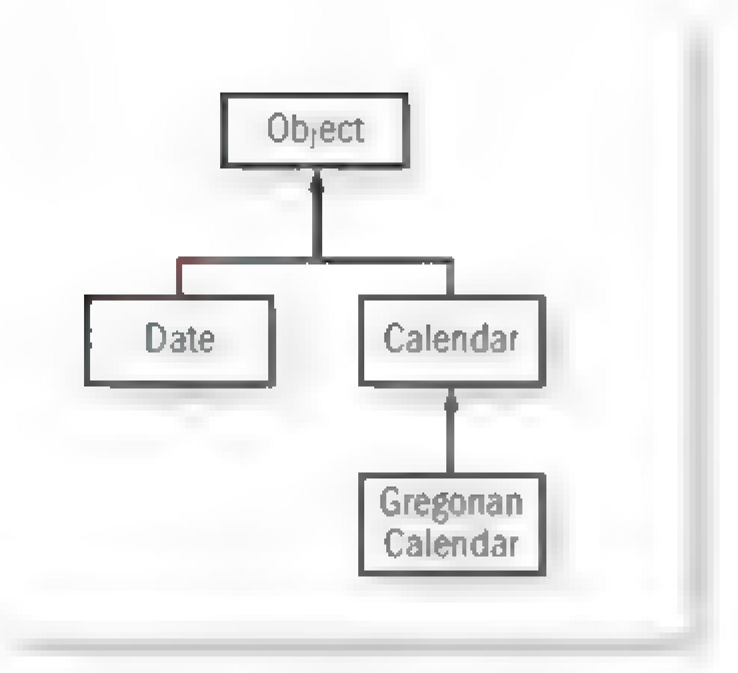


Figura 3. Jerarquía de Date y Calendar.

Revisemos el uso de esta clase en el ejemplo EjemploCalendar.class, disponible en el sitio **www.premium.redusers.com**.

java.lang.StringBuilder y java.lang.StringBuffer

En Java es muy sencillo concatenar strings, gracias al operador +. Lamentablemente, hacer uso intensivo de esta concatenación para generar texto es ineficiente. Recordemos que los strings son inmutables, y cuando queremos concatenarlos estamos creando todo el tiempo nuevos objetos que consumen mucha memoria.

```
String a = "Hola" + nombre + ". Buen dia!" +
pregunta + ". Hoy es " + new Date() + ".";
```

¿Cuántas cadenas de carácter se crean en el código anterior? ¿Cuatro, siete o trece? La respuesta es trece (al menos), ya que cada vez que usamos la concatenación estamos creando un String (aunque, en este caso, no nos interesen los intermedios).

Para poder realizar concatenaciones que no impacten en la velocidad y en la memoria se crearon dos clases especiales, StringBuilder y StringBuffer. Ambas permiten realizar las mismas operaciones de texto, con la diferencia de que la primera es una instancia que no puede ser utilizada desde distintos hilos de ejecución, mientras que la segunda (la más vieja de las dos implementaciones) es segura de usar en estos casos, aunque un poco más lenta.

Las operaciones más importantes son la **concatenación** (append) y la **inserción** (insert). También proveen métodos para reemplazar y borrar porciones de texto.

```
StringBuilder builder = new Stringbuilder();
builder
.append("Hola")
.append(nombre)
.append(". Buen dia!")
.append(pregunta)
.append(". Hoy es ")
.append(new Date())
.append(".");
String a = a.toString();
```

En este caso se han creado menos strings: ocho, contra los trece anteriores.

JAVA NIO

Ademas de la forma tradicional que tiene Java para manejar la entrada y salida de datos, existe lo que se conoce como **New I/O**. NIO define nuevas clases e interfaces para manejar de forma eficiente y asincrónica las lecturas y escrituras de datos. Si bien no es muy conocida y utilizada, existen productos que la usan, como el servidor **Jetty**.

Ejercicio: alternativa a java.util.Date

Vimos los problemas que tiene la clase Date y lo engorroso que es utilizar Calendar para manipular las fechas. El propósito de este ejercicio es modelar las fechas de forma distinta.

- Como primera premisa, tenemos que obtener un diseño donde las fechas sean inmutables y válidas desde su creación.
- Segundo, las entidades (conceptos) deben estar modeladas (por ejemplo, al pedirle a una fecha el mes, nos tiene que devolver un objeto que represente al mes en concreto y no un número que sea un índice arbitrario).
- Tercero, no tendremos en cuenta las distintas zonas horarias.
- Cuarto, separaremos el concepto de fecha del de hora.
- Por último, obviamente usaremos **TDD** para desarrollar.

Empecemos por modelar los años. ¿Qué podemos decir de los años? En principio, utilizamos un número para nombrarlos y no existe año cero. Luego, hay años que son bisiestos, donde febrero tiene un día más.

Pongamos estas ideas en tests:

```
@Test(expected=IllegalArgumentException.class)
public void testYearZero() {
        Year.number(0);
}

@Test public void testLeapYear() {
        final Year y2k = Year.number(2000);

        assertEquals(y2k.number(), 2000);
        assertTrue(y2k.isLeap());
        assertEquals(y2k.numberOfDays(), 366);
        assertEquals(y2k.february().numberOfDays(), 29);
}
```

El test para año no bisiesto es parecido, por lo cual lo obviamos aquí

Ya tenemos una buena aproximación de lo que podemos esperar inicialmente de un objeto que modele un año. Sigamos entonces con los meses. Tenemos doce meses: algunos de treinta días, otros de treinta y uno, y febrero, que, dependiendo del año, tiene veintiocho o veintinueve. Por lo tanto, con el mes solo no podemos saber cuántos días tiene ese mes: necesitamos un año en particular. Además, normalmente a los meses también se los identifica con su orden en el año.

```
@Test public void testMonth30() {
        final Month september = Month.september();
        final Year y2k = Year.number(2000);

        assertEquals(september.name(), "September");
        assertEquals(september.numberOfDaysIn(y2k), 30);
        assertEquals(september.monthNumber(), 9);
}
// Los tests para los meses de 31 días y para
febrero con 28 y 29 días siguen el mismo patrón
```

También podríamos haber optado por hacer que, en el caso de febrero, se arrojara una excepción, y no tener que pasar un año para conocer la cantidad de días que tiene el mes. El problema es que obligamos a todos los clientes a tener que manejar una excepción, lo cual es engorroso.

Una librería muy conocida en el mundo Java como alternativa al Date es Joda Time (**www.joda.org/joda-time**). Esta librería ofrece, además de la manipulación sencilla de fechas y horas, abstracciones de intervalos de tiempo, duraciones y distintos calendarios. Forma parte de un conjunto de librerías más grande y útil

Recordemos el test del año. ¿Por qué ahí pudimos saber cuántos días tenía febrero? Porque ese objeto, en realidad, representa un mes en un determinado año, y por lo tanto sabe cuántos días tiene. Es decir que los meses de un año determinado son distintos de los meses en general.

Ahora podemos proceder a modelar una fecha en particular, que puede ser vista como combinación de un año, un mes y un día (que representaremos con un número):

```
@Test public void testDateCreation() {
        final Year year1810 = Year.number(1810);
        final Month may = Month.may();
        final Date date = new Date(year1810, may, 25);

        assertEquals(date.year(), year1810);
        assertEquals(date.month(), may);
        assertEquals(date.day(), 25);
}
```

Esta es nuestra batería inicial de tests que nos servirán para construir nuestra solución al problema de las fechas.

Ahora, modelemos el tiempo. ¿Qué es el tiempo? Nosotros demarcamos el transcurso del tiempo utilizando horas, minutos y segundos (podemos seguir agregando precisión según necesitemos, pero dejaremos este problema para otro momento). Las horas pueden ser de cero a veinticuatro o de cero a doce si usamos **AM** y **PM**. Los minutos y segundos van de cero a cincuenta y nueve. Cada vez que una de las partes se pasa de su límite, la siguiente parte avanza en una unidad. En particular, cuando tengamos fecha y hora juntas, al pasarse la hora de su límite, el día tiene que avanzar.

Simplificaremos nuestra tarea pensando en un modelo sencillo donde el tiempo esté medido en horas, minutos y segundos.

Escribamos algunos tests:

```
@Test public void testCreationSuccess() {
        final Time teaTime = Time.hoursMinutesSeconds(17, 0, 0);

        assertEquals(teaTime.hours(), 17);
        assertEquals(teaTime.minutes(), 0);
        assertEquals(teaTime.seconds(), 0);
}
```

Y, por supuesto, los tests para los casos inválidos de creación:

```
@Test(expected=IllegalArgumentException.class)
public void testCreationFail_Hours() {
        Time.hoursMinutesSeconds(30, 0, 0);
}

@Test(expected=IllegalArgumentException.class)
public void testCreationFail_Minutes() {
        Time.hoursMinutesSeconds(10, 60, 0);
}

@Test(expected=IllegalArgumentException.class)
public void testCreationFail_Seconds() {
        Time.hoursMinutesSeconds(10, 0, -4);
}
```

Una librería que podemos utilizar como reemplazo de Date es **Date4J** **www.date4j.net**.
Ofrece fechas inmutables y una API muy sencilla para manipular fechas y horas

Con esto tenemos una base para poder trabajar con fechas y horas. Solo resta agregar los métodos que necesitemos y otros conceptos que puedan surgir (como las zonas horarias). Queda entonces en el lector la continuación del ejercicio y la obtención de un objeto que represente un instante (fecha y hora).

I/O

En esta sección, veremos las clases que nos permitirán acceder al filesystem de nuestra máquina y leer y escribir archivos. Estas clases se basan en la composición de **stream** (corrientes de datos) tanto para leer como para escribir. Cada tipo de stream ofrece un nivel de abstracción distinto (objetos, tipos primitivos, bytes, etcétera) y distintos medios (de la red, de un archivo, etcétera). Luego existen **writer** y **reader**, que ofrecen una visión de stream puramente de caracteres.

java.io.InputStream y su familia

La clase InputStream es la clase padre de todo stream de lectura. Define principalmente métodos para lectura de bytes (read), los cuales permiten leer un byte, o muchos, y copiarlos en un array.

COMPOSICIÓN DE STREAM

Una de las mejores características de diseño de la API de I/O es que se pueden ir componiendo distintos objetos para ir obteniendo distintas funcionalidades. Un ejemplo muy común es componer un **stream** de lectura con un BufferedInputStream para obtener una mejor performance, y luego con un ObjectInputStream para leer datos concretos (números, strings y objetos).

En caso de error, los métodos devuelven -1 como indicador de que no hay más información para leer, un signo de que se trata de un diseño antiguo y no orientado a objetos. Todos arrojan una IOExcepcion si hay algún error.

```
// Devuelve un byte o -1 si no hay más datos
int read()
// Devuelven -1 si no hay más datos
int read(byte [] datos)
int read(byte [] datos, int desde, int cuantos)
```

Por ejemplo, si queremos leer un archivo de nuestra computadora, tenemos que utilizar un tipo de stream para acceder al filesystem.

```
FileInputStream archivo = new FileInputStream('autoexec.bat');
byte [] datos = new byte[250];
try {
        Lee hasta los primeros 250 bytes del archivo
        archivo.read(datos);
} finally {
        // Nunca olvidarse de cerrar los streams
        archivo.close();
}
```

Es importante notar que siempre hay que cerrar los streams; si no lo hacemos, nos enfrentaremos a errores extraños en la aplicación.

java.io.OutputStream y su familia

No es una sorpresa que esta familia de clases se encargue de los streams de escritura. A diferencia de los InputStreams, este tipo de objetos define métodos para escribir (write) bytes a algún destino. Igualmente, todos los métodos de escritura pueden arrojar excepciones del tipo IOException en caso de error.

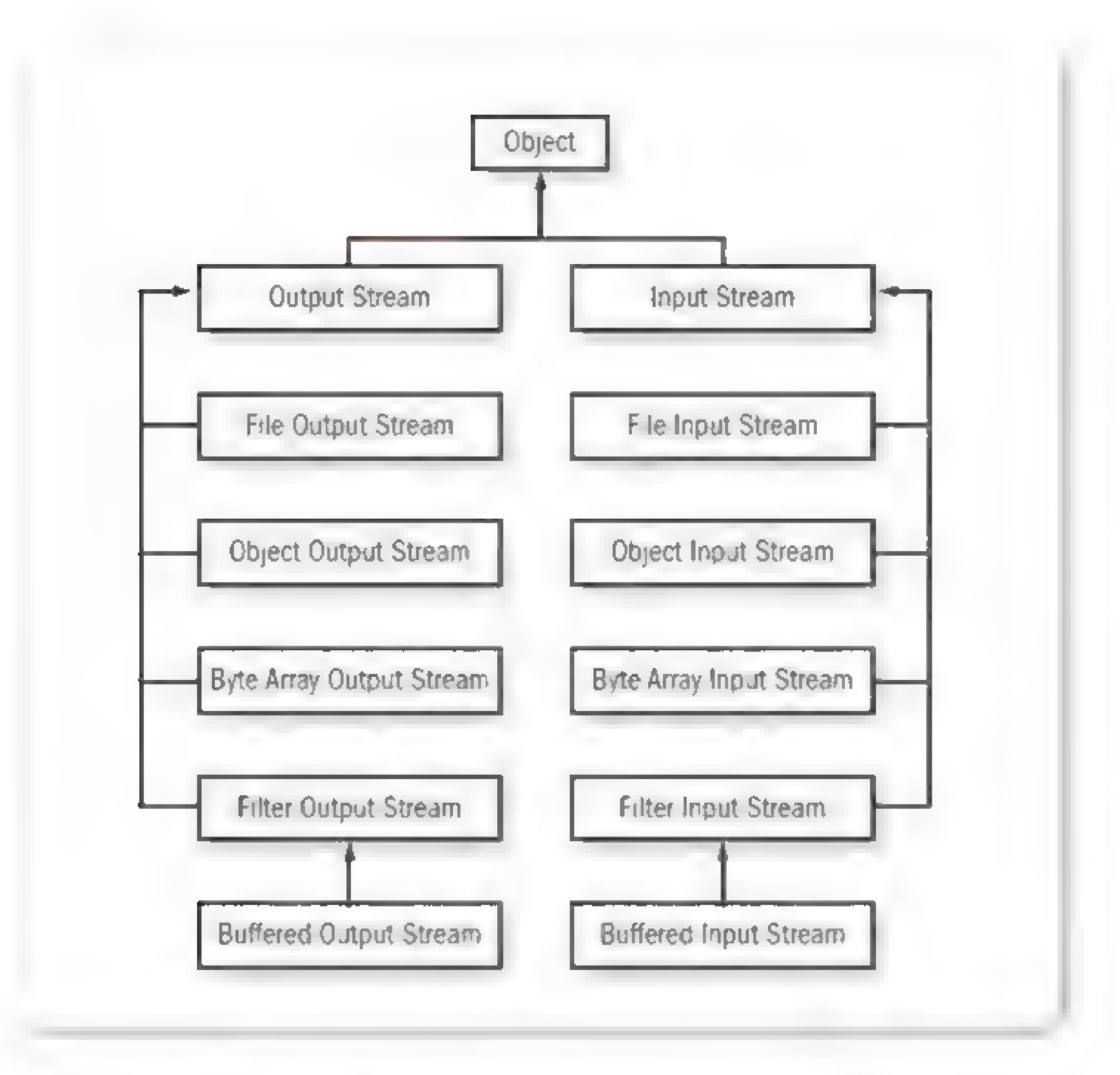


Figura 4. Jerarquia de los streams de lectura y escritura. Nótese la simetría en las jerarquías.

```
void write(byte)
void write(byte [] bytes)
void write(byte [] bytes, int desde, int cuantos)
  Fuerza la escritura de todos los datos pendientes
void flush()
```

De forma similar al ejemplo de la escritura, tratemos ahora de escribir un archivo:

```
FileOutputStream salida    new FileOutputStream("saludos.txt");
try {
        salida.write("hola a todos!".getBytes());
        salida.flush();
} finally {
        salida.close();
}
```

Recordemos cerrar también los streams de salida.

java.io.Reader y java.io.Writer

Esta familia de clases se asemeja a la de stream de lectura y escritura pero se focaliza en caracteres y strings en vez de hacerlo en bytes. El protocolo definido es similar al de InputStream y OuputStream, con la adición de algunos métodos. Veamos un segmento del protocolo definido por Reader.

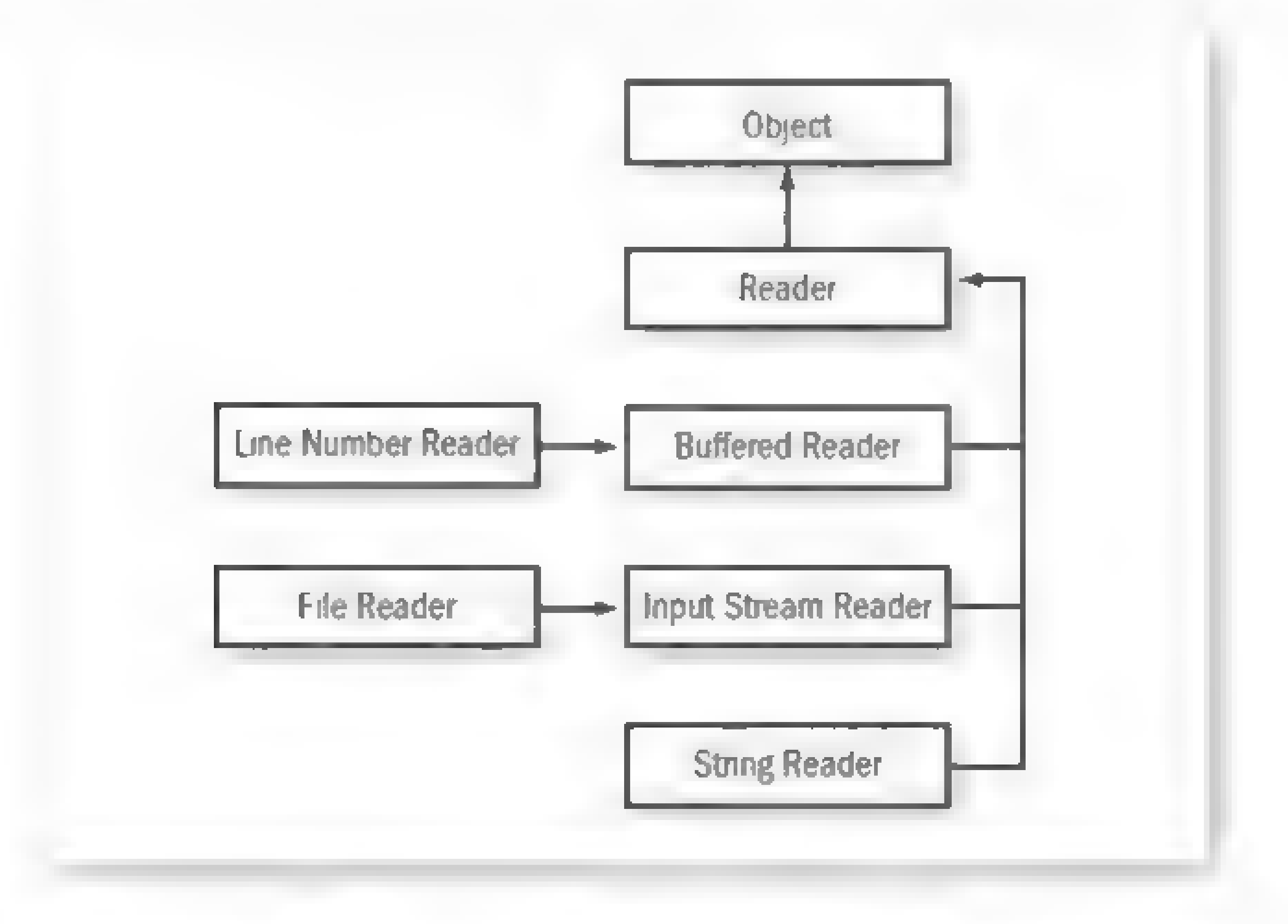


Figura 5. Jerarquia de los Reader.

```
// Devuelve un carácter o -1 si no hay más datos
int read()
// Devuelven -1 si no hay más datos
int read(char [] caracteres)
int read(char [] caracteres, int desde, int cuantos)
int read(CharBuffer contenedor)
```

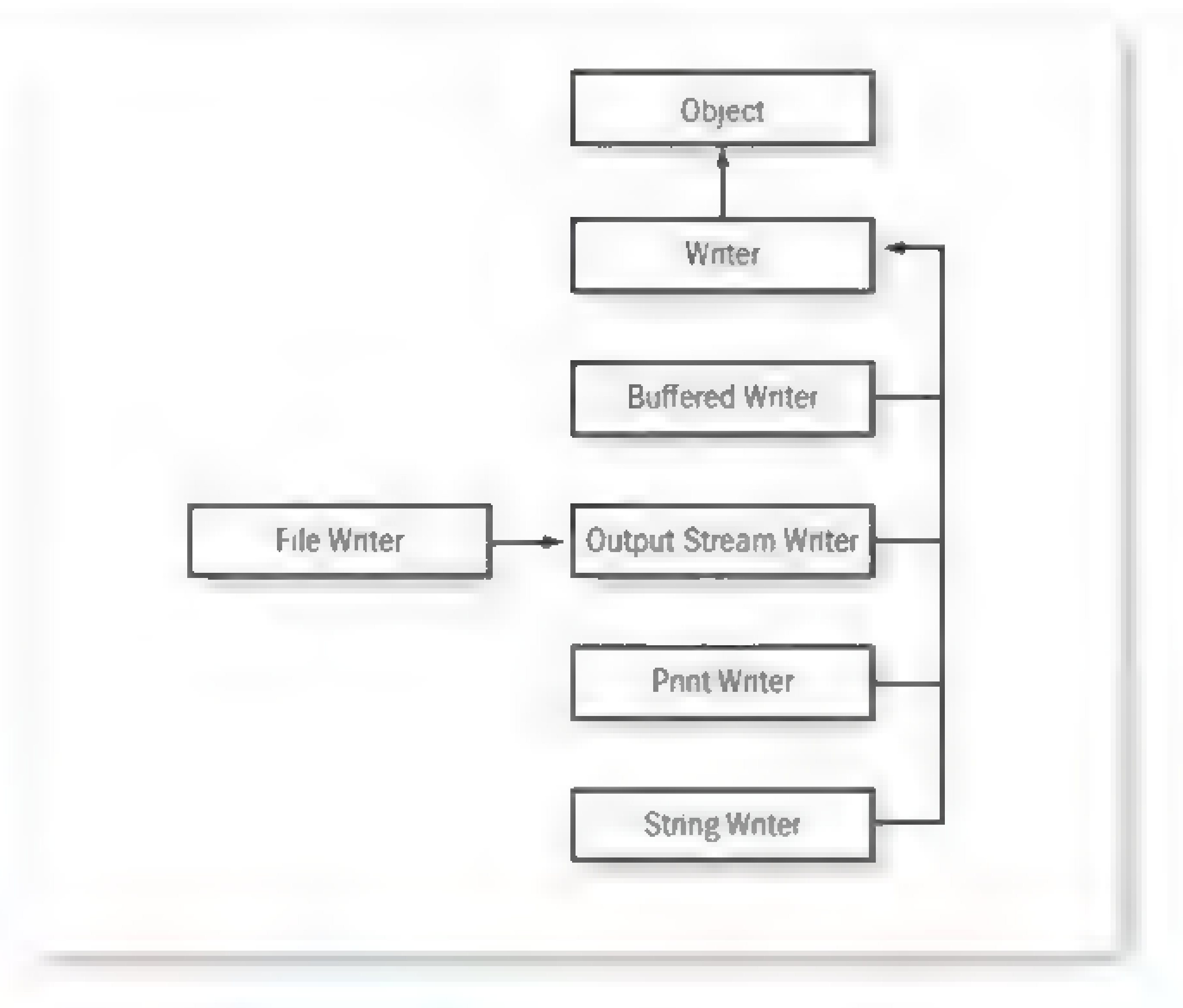


Figura 6. Jerarquía de los Writer.

Los streams ofrecen una abstracción a la transmisión de bytes y no están fijos a la red o a los archivos. Una opción común es obtener un **InputStream** a partir de un **String**.

Y ahora veamos algunos de los métodos declarados por Writer

```
Writer append(char caracter)
// Recordar que String hereda de CharSequence
Writer append(CharSequence secuenciaDeCaracteres)
// Extraño, ¿no? ¿Por qué no acepta un carácter?
void write(int caracter)
void write(String texto)
void write(char [] caracteres)
void close()
void flush()
```

Deberemos tener las mismas consideraciones que al trabajar con streams. Podemos encontrar este código de ejemplo en **www.premium.redusers.com**, bajo el nombre EjemploStream.class.

RESUMEN

Esta fue una breve introducción a las clases mas basicas que se ofrecen en la libreria estandar de Java, que ningun programador puede desconocer Las colecciones son parte fundamental de todo programa: por lo tanto, hay que conocer bien cuales son las capacidades de cada una de las implementaciones para poder seleccionar la mas adecuada a la tarea Las de manipular fechas y horas tambien son importantes, ya que un mal uso puede llevar a errores con los husos horarios dificiles de encontrar. Por último, conocer como transmitir y recibir información, ya sea de la red o de un archivo, es básico y merece un estudio mas profundo.

Actividades

TEST DE AUTOEVALUACIÓN

1 ¿Qué cuidados se debe tener al implementar equals?

2 ¿Qué cuidados se debe tener al implementar hashCode?

3 ¿Cuál es la diferencia entre StringBuilder y StringBuffer?

4 ¿Qué tipo de colecciones ofrece Java?

5 ¿Cuáles son las implementaciones más conocidas y usadas de cada una?

EJERCICIOS PRÁCTICOS

1 Finalice el ejercicio de las colecciones.

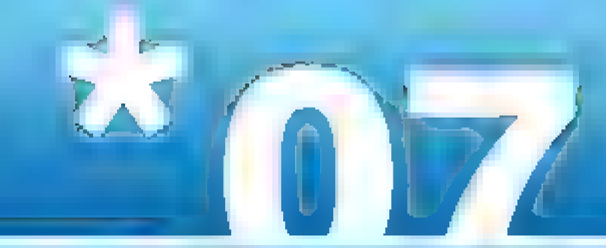
2 Agregue la interfaz Zippable, que ofrezca la funcionalidad zip.

3 Agregue la interfaz Filterable, que permite filtrar una colección dada una condición.

4 Agregue una interfaz que defina la capacidad para iterar una colección y actuar sobre sus elementos (forEach).

PROFESOR EN LÍNEA

Si tiene alguna consulta técnica relacionada con el contenido, puede contactarse con nuestros expertos: **profesor@redusers.com**.



Anotaciones

Muchos sistemas y librerías requieren una gran cantidad de información suplementaria al código para poder funcionar. Las anotaciones son un mecanismo para poder unificar toda esa información extra de código en una sola forma estandarizada. Esto permite no solo que los programadores eviten inventar formas de volcar la información extra, sino que también se ahorren la complicación de manipularla.

¿Qué son las anotaciones?

Durante la historia y evolución de Java se inventaron muchas formas de agregar información pertinente al código pero separada de él. Por ejemplo, los **JavaBeans** (objetos originalmente pensados para ser editados gráficamente) requerían la definición de una clase acompañante del tipo BeanInfo, donde se mantenía la descripción de bean. Los famosos **EJB** o *Enterprise Java Beans* requerían en sus versiones anteriores varias interfaces extras para poder ser usados en las aplicaciones de servidores, y también antiguamente se utilizaban comentarios en el código, que seguían cierto formato y utilizaban algunas herramientas (incluido el compilador) para funcionar.

Las anotaciones son artefactos que permiten especificar información extra respecto de una clase, método o variable, directamente en el código y de forma tal que pueden ser accedidas como un objeto por el sistema.

La presencia de anotaciones no afecta, inicialmente, la semántica del código, pero una herramienta o librería puede leer esta información y actuar modificando el comportamiento de los elementos anotados (es decir, aquellos afectados por las anotaciones).

En Java, las anotaciones se escriben junto con los modificadores del elemento en cuestión y su formato es el símbolo @ más el nombre de la anotación; opcionalmente, se escriben entre paréntesis una serie de argumentos.

JAVADOC Y XDOCLET

Desde sus comienzos y antes de que aparecieran las anotaciones, existían comentarios de código que servían para el mismo propósito. Estos se conocían como **xdoclet** y ofrecían una alta gama de tags para distintas funcionalidades. Se requería utilizar una herramienta extra que leía el código fuente y operaba en consecuencia a medida que iba encontrando los tags en los comentarios.

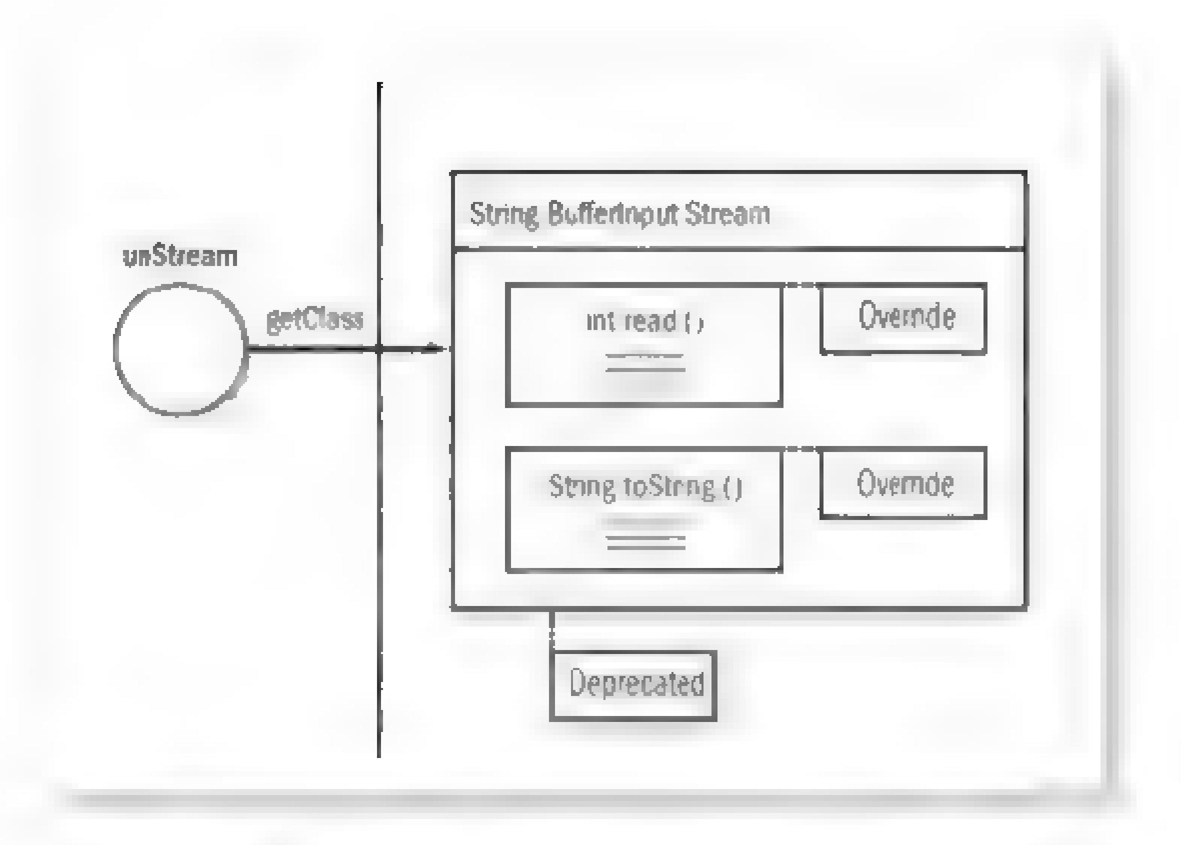


Figura 1. Las anotaciones no afectan la semántica de una clase, sino que agregan información a sus elementos.

Algunas anotaciones conocidas

Hemos estado utilizando estas anotaciones en los códigos que se encuentran en los capítulos anteriores. Repasemos algunas de ellas.

@Override

Esta anotación puede, en principio, parecer innecesaria, ya que el compilador sabe si estamos sobrescribiendo un método o no. Es conveniente utilizarla siempre, por varios motivos. Por ejemplo, si queremos sobrescribir un método y nos equivocamos al momento de escribir el nombre, el compilador nos alertará que no estamos sobrescribiendo ningún método conocido sino que la anotación será utilizada en tiempos de compilación. También es una forma de asegurarnos de que estamos implementando los métodos de una interfaz correctamente.

AUNQUE PUEDA PARECER INNECESARIO, ES CONVENIENTE USAR @OVERRIDE SIEMPRE

@Test

Otra anotación que utilizamos frecuentemente es @Test, usada en nuestros Tests Cases para indicar qué métodos representan pruebas unitarias.

Esta anotación puede aceptar algunos argumentos en su uso. Por ejemplo, si deseamos indicar que el test arroja una excepción, lo hacemos con el argumento expected:

```
@Test(expected=IOException.class)
```

También es posible indicar que la prueba debe tardar menos de una determinada cantidad de tiempo o, de lo contrario, fallará. Lo hacemos agregando el argumento timeout y especificando, a continuación, la cantidad de milisegundos que se debe esperar:

```
@Test(timeout=1000)
// Esperamos un segundo o falla
```

Ambos argumentos pueden usarse al mismo tiempo. La anotación @Test solamente puede decorar métodos que sean públicos y no devuelvan nada (void).

@Deprecate

Esta anotación es utilizada para indicar que un elemento (clase, interfaz, método, etcétera) no debe utilizarse más y que es posible que en futuras versiones sea removido. Generalmente se usa cuando un diseño es actualizado y se mantienen los elementos antiguos para mantener la compatibilidad hacia atrás.

Anteriormente, esta funcionalidad era ofrecida por un comentario que contenía el texto @deprecated, y se esperaba que el compilador alertara al programador de tales usos. @Deprecated puede ser utilizada para decorar cualquier elemento.

@SuppressWarnings

Esta anotación sirve para apuntarle al compilador que debe dejar de indicar alarmas del tipo especificado sobre un elemento determinado. Los tipos de alarma se especifican utilizando su nombre (en texto), y es posible indicar más de un tipo de alarma al mismo tiempo (con un array)

No está estandarizado qué tipo de alarmas debe soportar un compilador, pero las más comunes son las que se presentan en la tabla siguiente

ALARMAS	
ALARMAS	DESCRIPCIÓN
deprecation	Suprime las alarmas de uso de codigo deprecado
unchecked	Anula las alarmas de uso de llamadas o casteos no verificados (en tipos)
serial	Deshabilita las alarmas de las definiciones de tipo Serializable sin un serialVersionUID.
rawtypes	Deshabilita las alarmas relacionadas con el uso de tipos genéricos sin especificar los tipos.
finally	Anula las alarmas de los return dentro de los finally (ya que ocultan el return del try)
unused	Cancela las alarmas relacionadas con variables no utilizadas.

Tabla 1. Alarmas que podemos especificar.

Definición

La definición de una anotación no es muy distinta de la definición de cualquier interfaz, y la diferencia clave se encuentra en la utilización del símbolo @ antes de la palabra interface.

```
public @interface MiAnotacion {
    ...
}
```

Los argumentos que se le pueden pasar a una anotación se definen como si fueran métodos de la interfaz pero con algunas restricciones: no pueden tener parámetros o indicar que arrojan una excepción (utilizando throws); los tipos de retorno están restringidos a los tipos de datos primitivos (byte, char, short, int, long, float y double), cadenas de caracteres (String), clases (Class), otras anotaciones y arrays de los tipos anteriores; y los métodos pueden definir un valor por defecto, declarándolo con la palabra default seguida del literal apropiado.

```
public @interface Test {

        Class<? Extends Throwable> expected()
                                        default None.class;
         Declaramos el valor por default como 0 (nada)
        long timeout() default 0L;
}
```

En las versiones viejas de JUnit se utilizaba la convención de que los metodos que representaban pruebas unitarias eran aquellos cuyos nombres empezaban con el prefijo test Afortunadamente esto se dejó de usar, ya que un simple error de tipeo podía hacer que un test no fuera corrido y, por lo tanto, tuviéramos errores en el codigo.

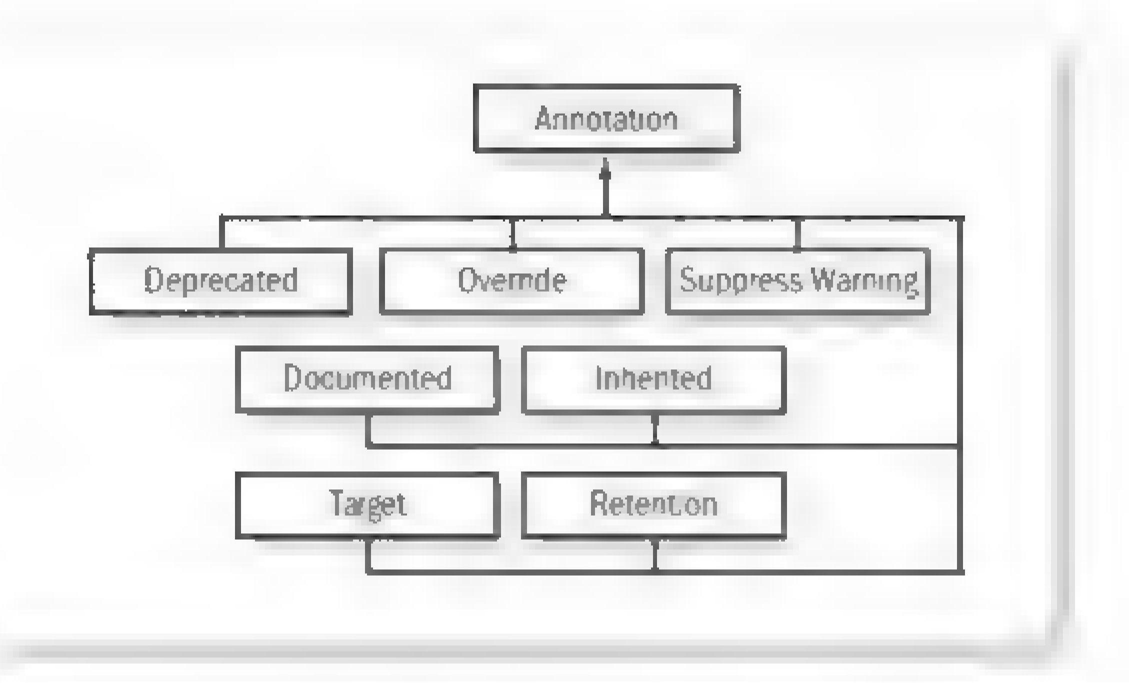


Figura 2. Jerarquía de las anotaciones en Java, formada por interfaces.

Cuando utilizamos un array, es posible indicar solamente un elemento y automáticamente será tomado como un array. Además, si nuestro parámetro se llama value, entonces, al momento de utilizarlo, no es necesario especificar su nombre y podemos pasar directamente el valor que queremos.

```
public @interface SuppressWarnings {
        String [] value();
}
// No tenemos que especificar value ni el array
@SuppressWarnings("rawtypes")
Map a = new HashMap();

  En este caso, sí los ponemos como array porque son más de uno
@SuppressWarnings({ "rawtypes", "unchecked"})
Map<String,String> b = (Map<String,String>) new HashMap();
```

Cuando utilizamos las anotaciones, los valores que usamos como argumentos deben ser literales (ya que son evaluados en tiempo de compilación). Las anotaciones que no tienen definido ningún método se conocen como **markers** (marcadores), ya que solo dan información por

su presencia (marcan el elemento decorado). Por su parte, las anotaciones que solamente tienen un argumento deberían llamarlo value, así no es necesario especificarlo cada vez que se las quiere utilizar.

Las anotaciones no pueden extender ninguna interfaz y tampoco pueden ser implementadas. Por lo tanto, una anotación no puede heredar de otra y tampoco formar jerarquías de tipos.

Al momento de definir las anotaciones, debemos indicar sobre qué elementos pueden ser aplicadas y cuál es el alcance que tienen. Para definir los elementos objetivos de la anotación, decoraremos la declaración de esta con la anotación @Target, que permite especificar los objetivos mediante un array de enumeraciones del tipo ElementType.

ElementType define los siguientes elementos: TYPE (clases e interfaces), CONSTRUCTOR (constructores), METHOD (métodos), FIELD (atributos), LOCAL_VARIABLE (variable local), PARAMETER (parámetro de un método), ANNOTATION_TYPE (declaración de una anotación, para definir **metaanotaciones**) y PACKAGE (declaración de un paquete).

También podemos definir el alcance de la anotación con @Retention, que especifica si la anotación es solamente para tiempo de compilación (SOURCE), si debe estar en el binario de la clase pero no necesariamente en tiempo de ejecución (CLASS) y si debe ser mantenida, incluso en tiempo de ejecución, para poder ser inspeccionada (RUNTIME). Esta enumeración corresponde al tipo RetentionPolicy.

NOMBRE DE LOS ARGUMENTOS

Ya vimos algunos consejos sobre los argumentos de las anotaciones, pero ellos no son todo. Siempre pensemos en cómo se usarán las anotaciones y de qué forma el cliente utilizará los argumentos. Debe ser bien claro cuál es el propósito de cada uno y cuáles son los valores permitidos. Tratemos de evitar los strings para otra cosa que no sea texto libre, a menos que sea necesario.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Test {
    ...
}
```

@Retention y @Target son llamadas metaanotaciones porque son anotaciones que decoran otras.

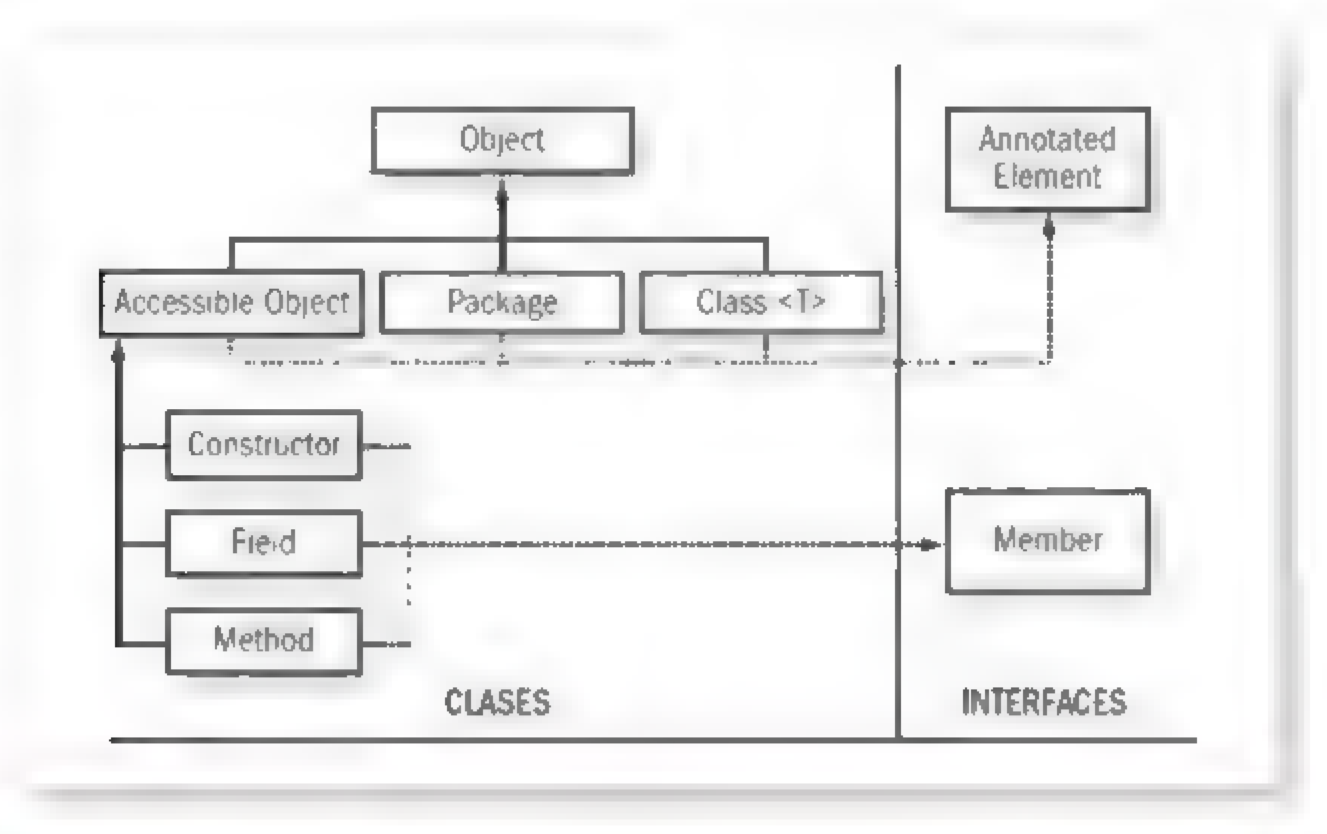
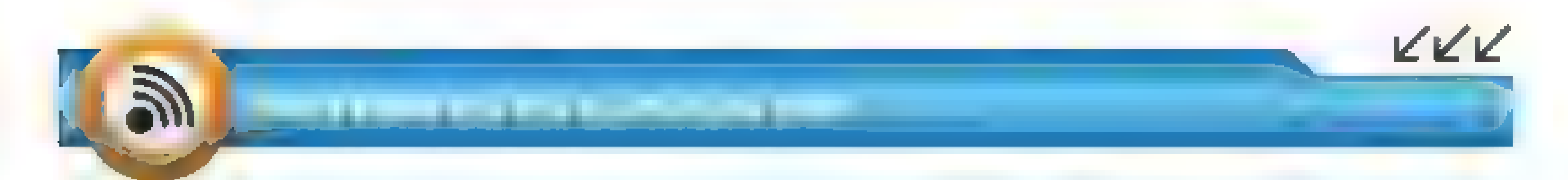


Figura 3. Jerarquía de los elementos que pueden ser decorados con anotaciones



Escomunconcebirlasanotacionescomounagranherramientay.porlotanto.quererutilizarlas para cualquier tarea. La realidad es que se deben utilizar en muy pocas situaciones y en casos específicos, ya que son elementos extraños. Cuando nuestro código tiene más anotaciones que código real, debemos preguntarnos si no estaremos diseñando mal.

Eclipse ofrece una opción para crear rápidamente una anotación básica. Para realizar esta tarea, debemos seleccionar File o presionar el botón New en la barra de herramientas, elegir la carpeta de los archivos fuentes e ingresar el paquete donde deseamos crearla. Luego tendremos que escribir el nombre de la anotación y presionar Finish.

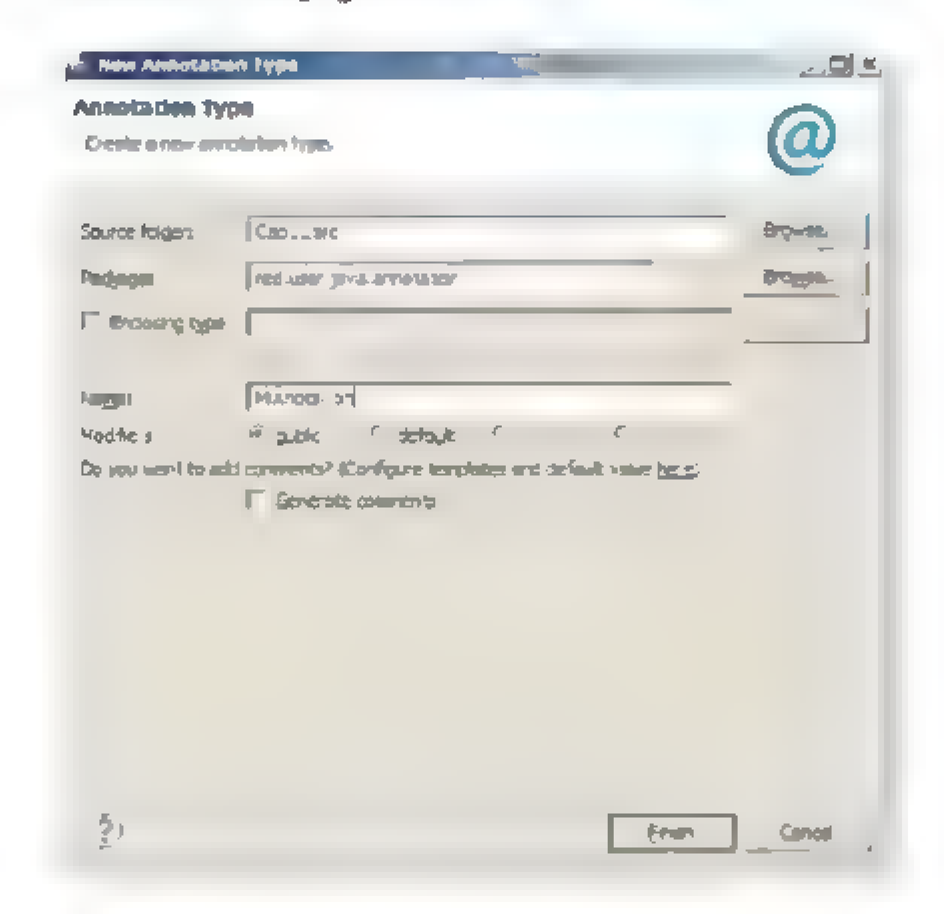


Figura 4. Ventana de creación de anotaciones en Eclipse.

Herencia de anotaciones

En principio, cuando anotamos una clase, no se considera que las subclases estén anotadas también, o sea, que la anotación afecte a las clases hijas. Por lo general, es el comportamiento deseado, pero si estamos seguros de que lo que queremos es que se propaguen las anotaciones en la jerarquía, debemos especificarlo.

Para especificarlo, al momento de definir la anotación que nos interesa que se herede, tenemos que marcarla con la anotación @Inherited. Esta permite que la anotación decorada con ella pueda ser heredada y transmitida a las clases hijas (heredar la anotación no significa que la clase hija declara el uso de la anotación).

Anotaciones en tiempo de ejecución

Ahora vamos a ver cómo podemos, desde nuestro código Java, acceder a la información declarada en las anotaciones de los objetos de nuestro sistema.

Tratando de seguir la filosofía de que todo es un objeto, Java provee ciertas clases que modelan a las clases, los métodos y demás elementos (como constructores, atributos, parámetros y paquetes). Sabemos que, para acceder a la clase de un objeto en particular, le debemos enviar el mensaje getClass() y, también, que podemos obtener el objeto clase si utilizamos un literal de clase como String.class. Una vez que tenemos acceso al objeto que representa la clase, le podemos pedir los demás elementos.

Las clases que representan a los distintos elementos implementan la interfaz AnnotatedElement, que ofrece un protocolo para obtener las anotaciones relacionadas con el elemento:

```
// Protocolo de AnnotatedElement

  Devuelve las anotaciones de un tipo determinado
<T extends Annotation> getAnnotation(Class<T> annontationClass);

  Devuelve todas las anotaciones del elemento (declaradas y heredadas)
Annotation[] getAnnotations();

  Devuelve todas las anotaciones directamente declaradas en el elemento
Annotation[] getDeclaredAnnotations();

  Responde si el elemento está anotado con un tipo de anotación en particular
boolean isAnnotationPresent(Class<? extends Annotation> annotationClass);
```

El lector alerta habrá notado el uso del tipo Annotation en el protocolo anterior. Esta interfaz es la representación del uso de una anotación en un elemento en tiempo de ejecución. Las anotaciones son interfaces, no tienen implementación concreta. Entonces, ¿dónde está definida la implementación?

La realidad es que el soporte de tiempo de ejecución crea una clase oculta que implementa la interfaz de nuestra anotación, y que implementa Annotation (al implementar nuestra anotación); de ahí que el tipo usado sea ese.

Veamos un ejemplo de cómo obtener la **metadata** de una clase:

```
public class AnnotationUnitTests {

 @Test(timeout=150)
 public void testXXX() {
/ Este método va a ser utilizado en el test siguiente
 }

 @Test
 public void testTestAnnotation() {

 Conseguimos la clase y le pedimos el método
  Method method =         this.getClass().getMethod("testXXX");

// Pedimos la anotación
  Test test = method.getAnnotation(Test.class);

// Ahora le pedimos a la anotación el timeout
  long timeout = test.timeout();

// Y ahora el error esperado
  Class<? extends Throwable> expected = test.expected();

// Verificamos que el timeout sea el correcto…
  assertEquals(timeout, 150);

// … y que no se espere ningún error
  assertEquals(None.class, expected);

 }

}
```

Podemos ver este código (AnnotationUnitTests class) en el sitio **www.premium.redusers.com** .

En el ejemplo, podemos observar cómo consultamos al objeto de nuestro test case por su clase, y luego a esta por un método del cual sabemos su nombre. Cuando obtenemos el método que nos interesa, le pedimos que nos devuelva la anotación del tipo @Test. Ahora tenemos en nuestras manos la instancia, un objeto, que representa la información que pusimos en nuestro código. Obtenemos, entonces, del objeto annotation, el timeout especificado y el error esperado (que, en este caso, no es ninguno), y validamos que lo que decimos sea correcto.

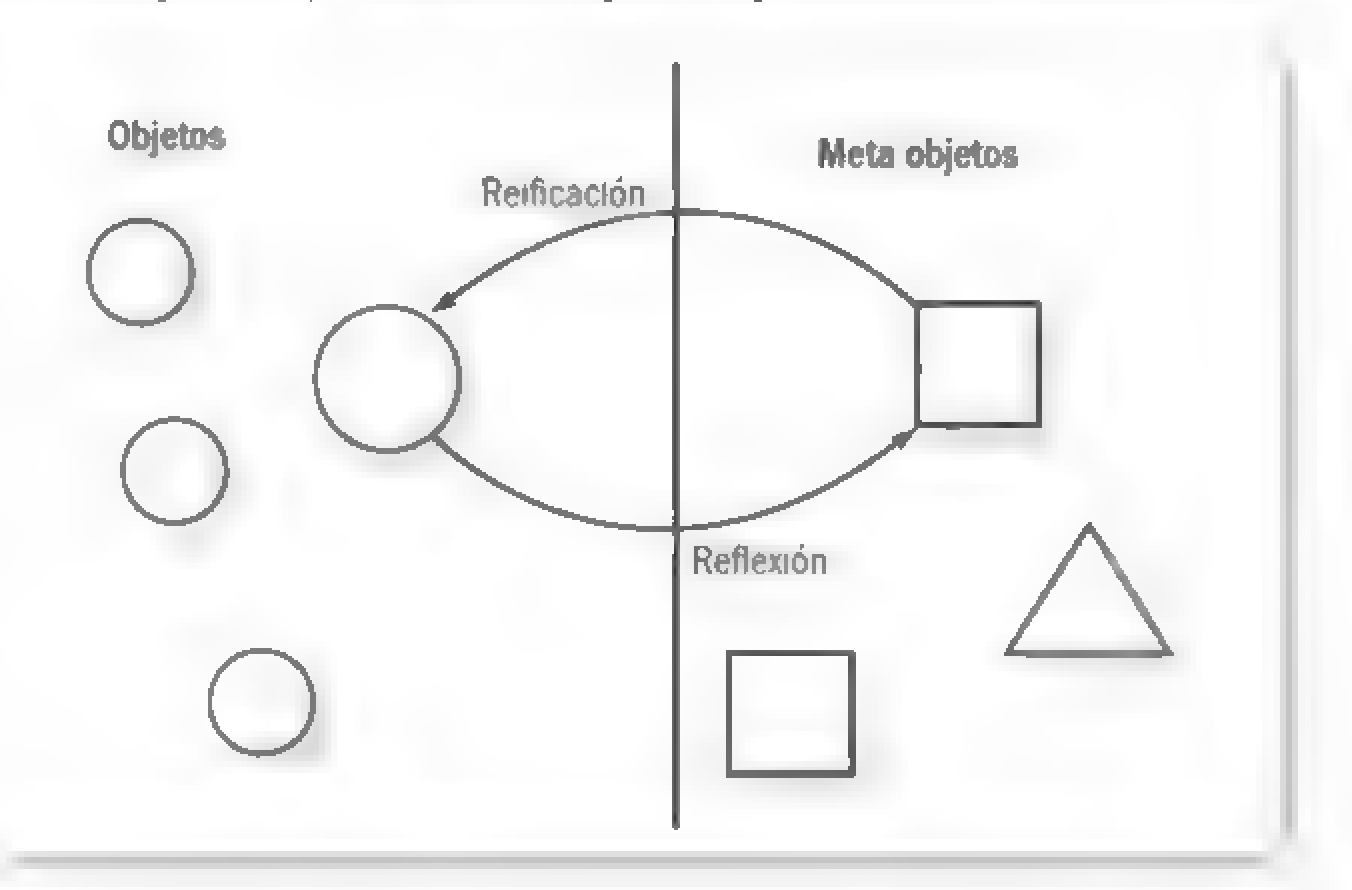


Figura 5. Visualización de objetos según pertenezcan o no a un nivel meta. El pase del nivel de dominio al nivel meta se da mediante un mensaje como getClass.

Jerarquizar anotaciones

Aunque las anotaciones sean interfaces, no se permite que extiendan ninguna otra anotación (ni interfaz). Esta es una limitación que solamente se hace evidente cuando queremos agrupar o jerarquizar distintas anotaciones. Supongamos que queremos crear un conjunto de anotaciones para

imponer restricciones en los argumentos de los métodos (que no sea nulo, que sea positivo, que sea una lista no vacía, etcétera). Cuando queramos acceder a estas anotaciones, deberíamos saber cuáles estamos buscando. Podemos saberlo por enumeración, conociendo todas las anotaciones de restricción que hayamos creado, pero, al agregar nuevas anotaciones, debemos cambiar esta búsqueda o idear algún mecanismo para notificar la existencia de una nueva restricción.

Otra forma de encarar esto es catalogar las anotaciones. No podemos hacerlo usando herencia, como dijimos, pero podemos decorarlas con otras anotaciones. De esta forma, podemos crear una metaanotación que identifique las restricciones y luego, al buscarlas, solo debemos asegurarnos de preguntar si están catalogadas como restricción o no. Así, las nuevas restricciones que creemos estarán naturalmente incluidas, sin trabajo extra.

Veamos cómo sería tal metaanotación:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface Restriccion {

}
```

REFLEXIÓN

El término **reflexión** se utiliza para indicar la capacidad de un sistema de mirarse a sí mismo, de evaluarse y, eventualmente, de modificarse. Específicamente, esto sucede cuando los metaobjetos del sistema pueden ser usados como objetos comunes y corrientes. En Java tenemos una clase que representa a las clases, otra que representa a los métodos y así con cada elemento de nuestro programa. Cuando pedimos la clase de un objeto, estamos pasando al lado de los metaobjetos.

Y, a continuación, observemos cómo se decoraría la restricción para que un argumento de un método no sea nulo:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.PARAMETER)
@Restriccion
public @interface NoNulo {

}
```

Al buscar las anotaciones, debemos preguntar por su categoría. Por ejemplo, veamos el código que sigue:

```
, Supongamos que annotation es una anotación de un
argumento de un método que nos interesa
if(annotation.annotationType()
          sAnnotationPresent(Restriccion.class)) {
  // Hacemos lo que tengamos que hacer
} else {
  // No hacemos nada con esta anotación
}
```

AYUDA DEL IDE

Una vez mas. la asistencia del **Eclipse IDE** es invaluable. Eclipse nos ayuda a completar los nombres de las anotaciones, y lo más importante: nos muestra cuáles son los argumentos que estas soportan y de qué tipo son. Recordemos que, para invocar el **autocomplete**, debemos presionar la tecla CTRL + barra espaciadora sobre la anotación y en el lugar donde van los argumentos.

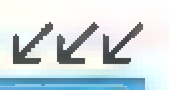
Esta forma de trabajo, combinada con lo que aprenderemos en el apartado siguiente, nos permitirá programar código altamente flexible y extensible para lidiar con las anotaciones.

Comportamiento en las anotaciones

Como hemos visto en la sección anterior, las anotaciones son, en definitiva, interfaces que no podemos implementar directamente. Entonces, ¿cómo podemos agregarles comportamiento y que no sean solamente contenedoras de información? Debemos crear alguna otra clase que contenga el comportamiento asociado a la anotación.

Para esto, lo mejor es mantener dicha anotación y dicha clase lo más cercanas y relacionadas posible. Si recordamos el capítulo sobre los distintos tipos de clases, seguramente recordaremos las clases internas, públicas y estáticas. Al ser interfaces, las anotaciones permiten los mismos elementos que estas, por lo que podemos crearles clases internas que sean las que operen sobre ellas y les provean cierto comportamiento manteniendo la cohesión.

Veamos un ejemplo. Supongamos que necesitamos una anotación para indicar que queremos memorizar los resultados de un método que es lento, a fin de que la próxima vez que lo utilicemos con los mismos resultados la respuesta sea instantánea (esto se conoce como **memoization**).



Java decidió no permitir extender las anotaciones mediante herencia debido a que, según el comité que aprueba los cambios en Java, haría más difícil la escritura de herramientas específicas para manipular las anotaciones. También aseguran que habría que mantener un sistema de tipos paralelo solamente para las anotaciones, que agregaría más complejidad tanto a la compilación como a la máquina virtual

Nuestro caso de prueba será el método para obtener el factorial de un número entero. Recordemos que el factorial (!) de un número entero **n** está definido como **1 si es 0** o como **n x (n - 1)!** en otros casos.

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ (n-1)! \times n & \text{if } n > 0. \end{cases} \qquad n! = \prod_{k=1}^{n} k$$

Figura 6. Dos definiciones de la función factorial.

```
public class Entero {
  ...
  @Memoize
  public Entero factorial() {
    if(this.esCero()) {
      return uno();
    }
    return this.multiplicadorPor(this.anterior().factorial());
  }
  ...
}
```

La mayoría de las anotaciones caseras se utilizan en tiempo de ejecución, pero también existe la opción de operar con ellas en tiempo de compilación. Java provee una herramienta llamada **APT** (**Annotation Processing Tool**), que permite enganchar nuestro código y dejarnos generar código, y elevar alarmas o errores de compilación que dependen de las anotaciones encontradas en los códigos fuentes.

Ya hemos anotado el método factorial para ser memorizado. De esta forma, cuando queramos el factorial de 4 (4!), la primera vez se calcularán los factoriales de 1, de 2, de 3 y de 4 y quedarán memorizados. La próxima vez que se pida el factorial de alguno de ellos, este será obtenido sin recalcular nada y sin invocar el factorial de otros números.

Ahora nos toca crear la anotación, que en principio solamente es un **marker**, así que no le definiremos ninguna propiedad. Obviamente, nuestra anotación debe tener alcance de tiempo de ejecución, ya que nos interesa modificar el comportamiento mientras se ejecuta la aplicación y solamente queremos recordar métodos particulares.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Memoize {

}
```

Luego necesitamos codificar todo lo necesario para recordar que cuando se llama a determinado método de cierta clase, con una lista de parámetros determinados, debemos devolver el valor recordado o invocar el método, recordar el resultado para después y devolverlo. Todo este código debería ser cercano a @Memoize; por lo tanto, vamos a ponerlo en una clase interna estática y pública.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Memoize {

  public static class Apply {
    ...
    public static <T> T to(T target) {
```

```
// Acá va el código necesario para devolver un
objeto con el comportamiento de memorización
    }
    ...
  }
}
```

Una vez que tengamos esto, podemos usarlo para crear instancias que puedan recordar los mensajes enviados a ellas y devolver la respuesta instantáneamente en vez de volver a ejecutar el método asociado. Esta es una práctica muy común (también conocida como **caching**), usada para mejorar la velocidad de código crítico.

Notemos que el hecho de que se recuerden los resultados es totalmente ortogonal a cómo es el objeto al que queremos aplicarle esta funcionalidad y cuál es su comportamiento. No programamos este tipo de funcionalidad directamente en cada método que queremos acelerar, sino que lo hacemos una vez y luego lo aplicamos a cada caso.

Tanto la anotación como el comportamiento asociado a ella están definidos juntos, por lo que no tenemos que adivinar cómo se utilizan y, de estar separados, dónde se encuentran.

SOBRE MEMOIZATION Y CACHING

Recordar resultados pasados para no tener que realizar el trabajo de obtenerlos nuevamente es una técnica ampliamente utilizada. Desde los servidores web, que memorizan páginas completas, hasta los números factoriales, como en nuestro ejemplo. Hay que mencionar que no es fácil implementar correctamente estos mecanismos, ya que debemos regular cuánta memoria estamos usando y su liberación.

Usos de las anotaciones

Hay infinidad de aplicaciones, librerías y **frameworks** que hacen uso de las anotaciones. Ya sea para proveer cierta funcionalidad o para ampliar una funcionalidad base, los usos de las anotaciones son vastos.

Validaciones

Java propone un especificación donde aparecen varias anotaciones para validar objetos (tanto los atributos de un objeto como los parámetros de un método). Algunas de las anotaciones definidas permiten declarar restricciones sobre las referencias y los objetos, tales como no ser nulas, estar entre cierto rango de números, tener una determinada longitud, o si una fecha debe ser en el futuro o en el pasado. También permiten declarar precondiciones, poscondiciones e invariantes de métodos y clases, práctica conocida como **diseño por contrato**. Además de la especificación, existen varias librerías que, dadas esas anotaciones de una forma u otra, realizan estas validaciones.

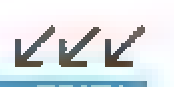
> EXISTEN, ADEMÁS, VARIAS LIBRERÍAS QUE PUEDEN REALIZAR ESTAS VALIDACIONES

UÉ ES UN FRAMEWORK?

Un **framework** es un conjunto de elementos reutilizables que conforman un sistema con un propósito determinado. Tal sistema está incompleto, en el sentido de que tiene "huecos" que completaremos con nuestro codigo, adecuando el comportamiento general a nuestras necesidades. La característica principal, frente a una librería, es que es el código del framework el que tiene el control total, y no nosotros.

```
class Persona {
        @NotNull @Length(min  3)
private String nombre;

@Min(0) private int edad;

...

boolean esHermanoDe(@NotNull @Valid otraPersona)
{
        ...
}

...

  Al usar esHermanoDe con null debería arrojar una excepción
juan.esHermanoDe(null)

}
```

En general, las restricciones y validaciones deberían estar encapsuladas en los objetos correctamente modelados. Por ejemplo, el nombre no debería ser tan solo un String, sino un objeto nombre propiamente dicho y, dentro de él, debería contener las validaciones necesarias para que las instancias sean válidas.



Oval (**http://oval.sourceforge.net**) es un proyecto dedicado a la creación de anotaciones para la validación de clases y métodos. Define una amplia variedad de anotaciones de restricciones sobre los atributos, parámetros y resultados, permite definir nuestros propios validadores y también la posibilidad de escribir código validador directamente en las anotaciones

Inyección de dependencias

Los objetos dependen de otros objetos con los cuales colaboran para realizar un objetivo específico. Un problema es cómo enlazar los objetos entre sí. Hay casos en que los colaboradores de un objeto son externos a él, donde son pasados en el momento de la construcción o mediante un método setter. En otros casos, son internos al objeto, como, por ejemplo, una colección para mantener ciertos objetos (externos o no). Esta colección seguramente será creada por el mismo objeto pero, en los otros casos, ¿quién se encarga de inicializar los objetos y dependencias y de pasárselos al que los necesita?

Podemos escribir este código que pega unos objetos con otros, pero existen varios proyectos que solucionan el problema, utilizando anotaciones para configurar cómo debe realizarse la inicialización de cada objeto y de sus dependencias.

```
public class VideoClub {

  @Autowired
  private CatalogoDePeliculas catalogo;

  ...
}
```

Este framework de inyección de dependencias creado por el gigante **Google** (**https://github.com/google/guice**) permite la rápida y sencilla configuración de nuestros objetos. **Guice** se encarga de manejar todo el código que pega a los objetos entre sí, haciéndose cargo de su creación, configuración y cuidado. La idea es dejar de encargarse uno mismo de crear las instancias de los colaboradores, para delegar esta responsabilidad a otra parte.

En el ejemplo, el catálogo de películas será pasado automáticamente al objeto del videoclub en el momento de creación.

Serialización

Se dice que un objeto se serializa cuando es transferido fuera de la máquina virtual, ya sea a un archivo, a otra máquina virtual por la red o a un formato distinto, como un archivo **XML**. Si bien Java ofrece mecanismos para serializar semiautomáticamente objetos, lo hace de forma binaria.

Hoy, sobretodo en la Web, se utilizan formatos basados en texto como XML o **JSON**. Por ejemplo, en el caso de XML, la traducción entre objetos y XML es un tanto ambigua, ya que algunos atributos pueden ser elementos en el XML. Para poder configurar tal traducción, muchas herramientas hacen uso de las anotaciones. Veamos el ejemplo de una agenda:

```
@Root
public class Agenda {
        ...
        @ElementList
        private List<Contacto> contactos;
        ...
}
```

Hibernate es un amplio y conocido proyecto de **ORM** en Java, que podemos encontrar en **www.hibernate.org**. Ofrece funcionalidad de punta a punta en el desarrollo de soluciones que se comunican con bases de datos, desde la creación de las tablas hasta el mapeo de los objetos de estas, soportando casi todas las bases existentes. Utiliza anotaciones para configurar el mapeo.

Creamos la clase contacto:

```
@Root
public class Contacto {
        ...
        @Attribute
        private String nombre;

        @Element
        private Direccion direccion;

        private List<String> telefonos;
        ...
}
```

Creamos la clase dirección:

```
@Root
public class Direccion {
        ...
        @Element
        private String calle;

        @Element
        private String numero;
        ...
}
```

Si ahora queremos serializar un objeto del tipo Agenda, obtendremos el siguiente XML:

```
<agenda>
  <contactos>
    <contacto nombre="Juan Perez">
      <direccion>
        <calle>Lavalle</calle>
        <numero>123</numero>
      </direccion>
      <telefono>134-23456</telefono>
    </contacto>
    <contacto nombre="Armando Tranvias">
      <direccion>
        <calle>De las flores</calle>
        <numero>56</numero>
      </direccion>
      <telefono>456-4343</telefono>
      <telefono>456-4344</telefono>
    </contacto>
  </contactos>
</agenda>
```

En este capítulo hemos aprendido que las anotaciones permiten agregar información al código de forma sencilla y práctica. Además, ofrecen la posibilidad de inspeccionarlas tanto en tiempo de ejecución como en tiempo de compilación. Pero, antes de lanzarnos a utilizar las anotaciones, debemos preguntarnos si son la solución correcta a nuestro problema, o si debemos mejorar nuestro diseño y bajar la información del nivel meta al nivel de los objetos de nuestro dominio.

Actividades

TEST DE AUTOEVALUACIÓN

1 ¿Qué son las **anotaciones**?

2 Nombre algunas de las anotaciones que afectan al compilador.

3 ¿Es posible que un método sobrescrito herede las anotaciones del método original?

4 ¿Puede una anotación extender otra?

5 ¿Cuáles son los tipos permitidos como propiedades de las anotaciones?

EJERCICIOS PRÁCTICOS

1 Cree una anotación para indicar que no se aceptan nulos, llamada @NoNulo.

2 Cree una clase de prueba como, por ejemplo, Persona, y aplique la anotación a sus atributos y a los argumentos de los métodos y constructores.

3 Agregue un método estático que permita instanciar objetos de la clase anterior y utilícelo en lugar de hacer new.

4 Agregue un método que valide el estado del objeto (de sí mismo) basándose en las anotaciones.

PROFESOR EN LÍNEA

Si tiene alguna consulta técnica relacionada con el contenido, puede contactarse con nuestros expertos: **profesor@redusers.com**.

Técnicas y diseño

A lo largo de los capítulos anteriores hemos aprendido cómo está formado y qué nos ofrece Java. Es tiempo de aprender a utilizar esos conocimientos y herramientas de forma adecuada: debemos saber de qué manera combinar los elementos del lenguaje para construir programas claros, extensibles y flexibles. Para esto, conoceremos ciertos conceptos de diseño que nos serán muy útiles.

Java Beans y creación de objetos

Es común encontrar objetos que solamente tienen **getters** y **setters** (tema que podemos repasar en el **Capítulo 3**): estas clases se denominan **Java Beans**. Su nombre no tiene un procedimiento bien definido, ya que históricamente se vienen utilizando con esa nomenclatura.

Estos objetos no tienen un comportamiento definido, sino que solamente son contenedores de datos. En sí mismo, esto no es un problema, pero su comportamiento, en realidad, se encuentra en otros objetos que solamente tienen métodos para manipular los beans. Por lo tanto, se presenta una situación en la cual hay dos objetos: uno para contener la información y otro para operar sobre esta; así, básicamente, no estamos utilizando el paradigma orientado a objetos sino programando de vuelta en C.

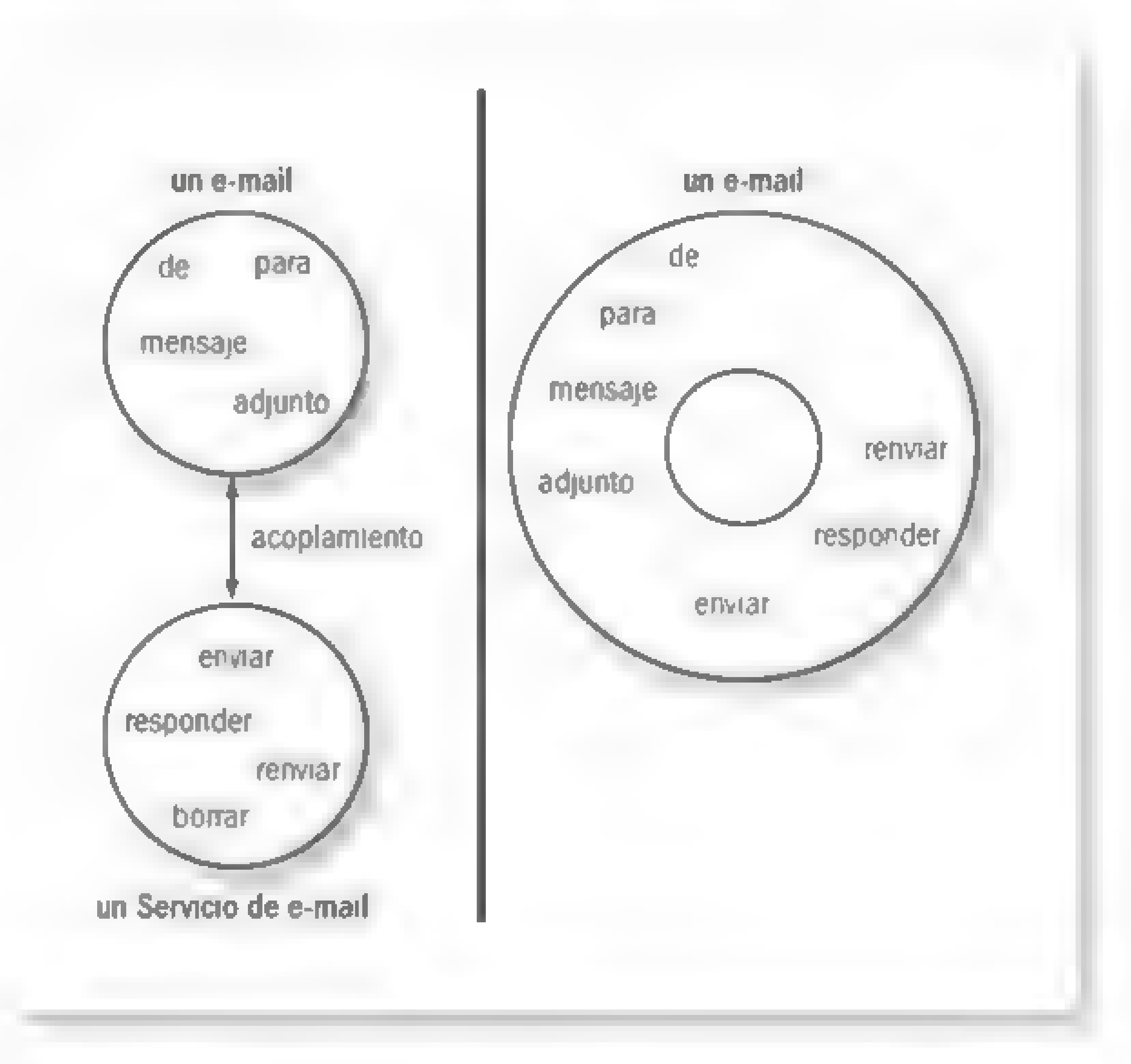


Figura 1. Acoplamiento entre el objeto **Java Bean** y el objeto que tiene la lógica para manipularlo.

Recordemos que los objetos encapsulan su estado (información) y operaciones (métodos) en un solo lugar. Dividir esto en dos es crear problemas innecesarios. Si encontramos clases que solo definen atributos con setters y getters, debemos buscar las clases que operan sobre la primera y tratar de mover la funcionalidad de una clase a otra; así, mantendremos juntos el estado y la funcionalidad.

Por ejemplo, esta sería una hipotética clase que modela un e-mail. Es una clase "boba", ya que no hace nada más que contener datos:

```
public class Mail {
	private String to;
	private String from;
	private String body;

	public String getTo() {
		return to;
	}
	public void setTo(final String to) {
		this.to = to;
	}
	public String getFrom() {
		return from;
	}
	public void setFrom(final String from) {
		this.from = from;
	}
	public String getBody() {
		return body;
	}
	public void setBody(final String body) {
		this.body = body;
	}
}
```

A continuación, veamos la clase que permite manipular y operar un e-mail. Sin esta, la anterior es inútil:

```
public class MailService {
        public void enviarMail(final Mail mail) {
                ...
        }

        public void reenviarMail(final Mail mail, final String destinatario) {
                ...
        }

        public void responderMail(final Mail mail, final String respuesta) {
                ...
        }
        ...
}
```

¿Tiene sentido mantener dos cosas cuando podríamos mantener solo una? Si sabemos que una clase es inútil, ¿para qué la tenemos? Estas son algunas de las preguntas que debemos hacernos cuando nos enfrentamos a este tipo de diseños.

PROGRAMANDO CON ESTRUCTURAS

Este estilo de programación, que utiliza objetos que solo contienen datos y, luego, en otro objeto o clase, tiene métodos para manipular y utilizar estos objetos, viene de **C** y de otros **lenguajes procedurales** . En estos lenguajes no hay objetos, y priman los procedimientos que usan datos.

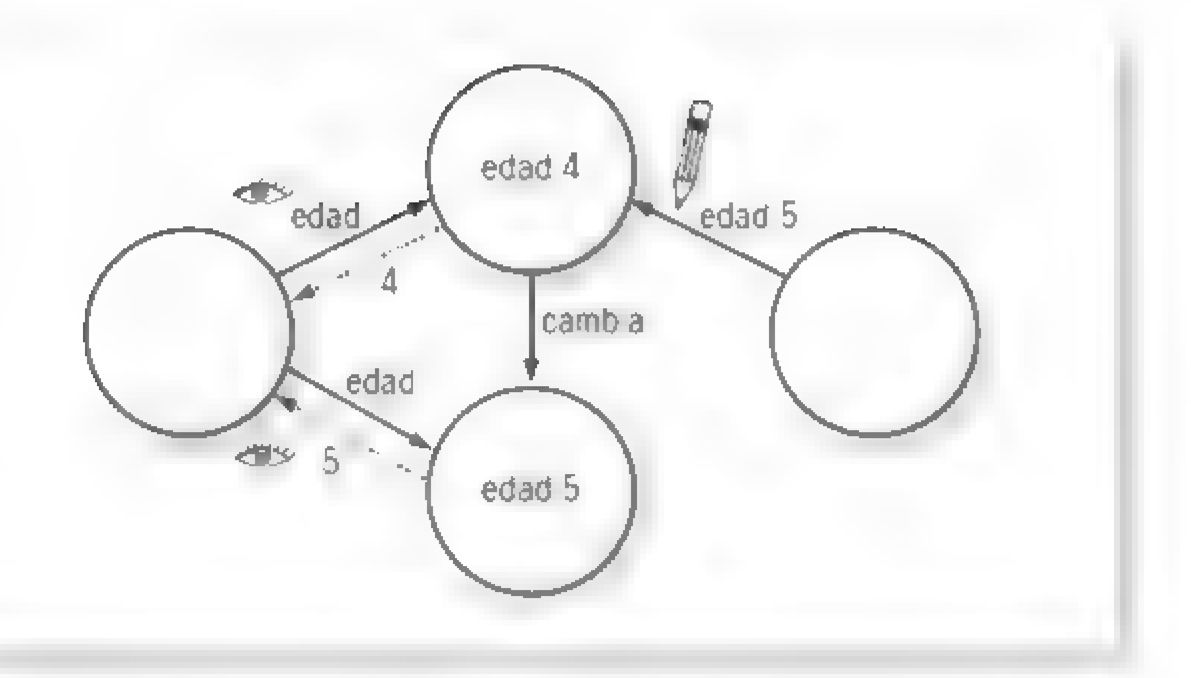


Figura 2. Los objetos mutables pueden producir errores.

Otra característica pobre de los objetos puramente de datos se halla en que, casi en su mayoría, son mutables, es decir que pueden cambiar en cualquier momento. Si bien esto puede ser lo que busquemos, en general, los objetos del mundo real son inmutables. Supongamos que tenemos el objeto fecha 10 de febrero de 2020: si pudiéramos cambiar la propiedad dia por 20, este dejaría de modelar la fecha inicial y el resto del código seguiría creyendo que es 10. Así, se producen errores extremadamente difíciles de encontrar y arreglar. Sería más fácil si desde el principio el objeto fecha no pudiera ser modificado. Si queremos cambiarla, simplemente creamos otra fecha, sin afectar al resto del código que depende de la fecha original.

BENEFICIOS DE LA INMUTABILIDAD

Los objetos inmutables tienen varios beneficios: son fáciles de crear, probar y usar; son **thread safe** (no requieren sincronización); no requieren ser copiados defensivamente cuando son atributos y se los devuelve en un método, son excelentes candidatos para estar en un Set o como claves de un Map, su invariante de clase se valida una sola vez al construirlos y nunca están en un estado inválido.

Esto es muy importante, especialmente en ambientes de muchos procesos en paralelo, como una aplicación web, que atiende varios clientes simultáneamente. Además, al hacer los objetos inmutables, nos protegemos de devolver un colaborador interno como respuesta de un método y evitamos que un cliente modifique tal objeto. Como regla general, los objetos creados deberían ser inmutables.

Para lograr objetos inmutables en Java debemos utilizar el modificador final en los atributos e inicializarlos utilizando el constructor (o, directamente, en la definición de los atributos).

```
public class Punto {
  private final double x;
  private final double y;

  public Point(final double x, final double y) {
    this.x = x;
    this.y = y;
  }

  public double x() {
    return x;
  }

  public double y() {
    return y;
  }
}
```

En objetos que requieran de mayor inicialización, podemos utilizar setters privados y tratar de usarlos solamente desde un único punto, el constructor

No solo debemos tratar de controlar los cambios que pueden afectar a un objeto determinado, sino que también tenemos que cuidarnos de que solamente se creen objetos que sean válidos. Reflexionemos: ¿nos interesa

tener una fecha inválida? ¿De qué nos sirve? ¿Hay fechas inválidas en la realidad? Por definición, una fecha es válida siempre; si no, no existe. Esta restricción también tiene que ser modelada y, por lo tanto, debemos asegurarnos de crear objetos correctos desde el inicio de su vida.

Supongamos que dejamos que se creen fechas sin inicializar y que podemos ir cambiando las propiedades de día, mes y año:

```
Fecha fecha = new Fecha();
fecha.setDia(29);
fecha.setMes(2);
fecha.setAño(2020);
// ERROR ¡fecha no válida!
// Arreglémosla
fecha.setAño(2012);
// Ahora sí existe
// También podríamos haber hecho…
Fecha.setDia(28);
```

La validación se tiene que realizar en cada paso, ya que ante cualquier acción puedo tener una fecha inválida. Además, hasta que no está inicializada, la validación no tiene sentido. Esto implica mucha lógica, que tiene que estar respaldada por mucho código, solo para tener fechas válidas; podríamos conseguirlo tan solo inicializando y validando en el constructor.

```
// ERROR fecha inexistente
Fecha fecha = new Fecha(29, 2, 2011);
// Fecha correcta
Fecha fecha = new Fecha(29, 2, 2012);
```

En el constructor, hacemos las validaciones pertinentes:

```
public class Fecha {
  ...
  public Fecha(final int dia, final int mes, final int año) {
        // Inicializo las variables y verifico
    if(!valida())
      throw new FechaInvalidaException();
  }
  ...
}
```

Incluso sería mejor modelar correctamente las entidades de mes y año, y las relaciones entre ellas.

Lo importante de esta sección es que debemos restringir las modificaciones a los objetos tanto como podamos y de acuerdo a cómo se toman esas modificaciones en el dominio del problema. Evitaremos, en lo posible, los setters y las clases bobas que solo contienen datos. Finalmente, trataremos de darles semántica a los constructores o, si tenemos varias formas de creación para un mismo tipo de objeto, utilizaremos métodos estáticos.

Inyección de dependencias e inversión de control

En el capítulo anterior mencionamos algunos **frameworks de inyección de dependencias**. Veamos ahora, con mayor detalle, de qué se trata este concepto y cómo se relaciona con otro: la **inversión de control.**

La inversión de control trata de distribuir las responsabilidades de un sistema en varios objetos, de modo que cada uno tenga una responsabilidad clara, bien definida y con un único propósito. Este concepto está asociado generalmente a la inicialización de colaboradores (de ahí que se lo relacione solamente a la inyección de dependencias), pero aplica a todo

tipo de responsabilidad. Por ejemplo: si tenemos un objeto con varios atributos y varios métodos, y solo una parte de esos métodos actúa sobre una parte de los atributos. Claramente, este objeto está teniendo dos responsabilidades y podríamos dividirlo fácilmente en dos objetos, donde cada uno, al ser responsable de su funcionalidad, colabora para proveer la funcionalidad original.

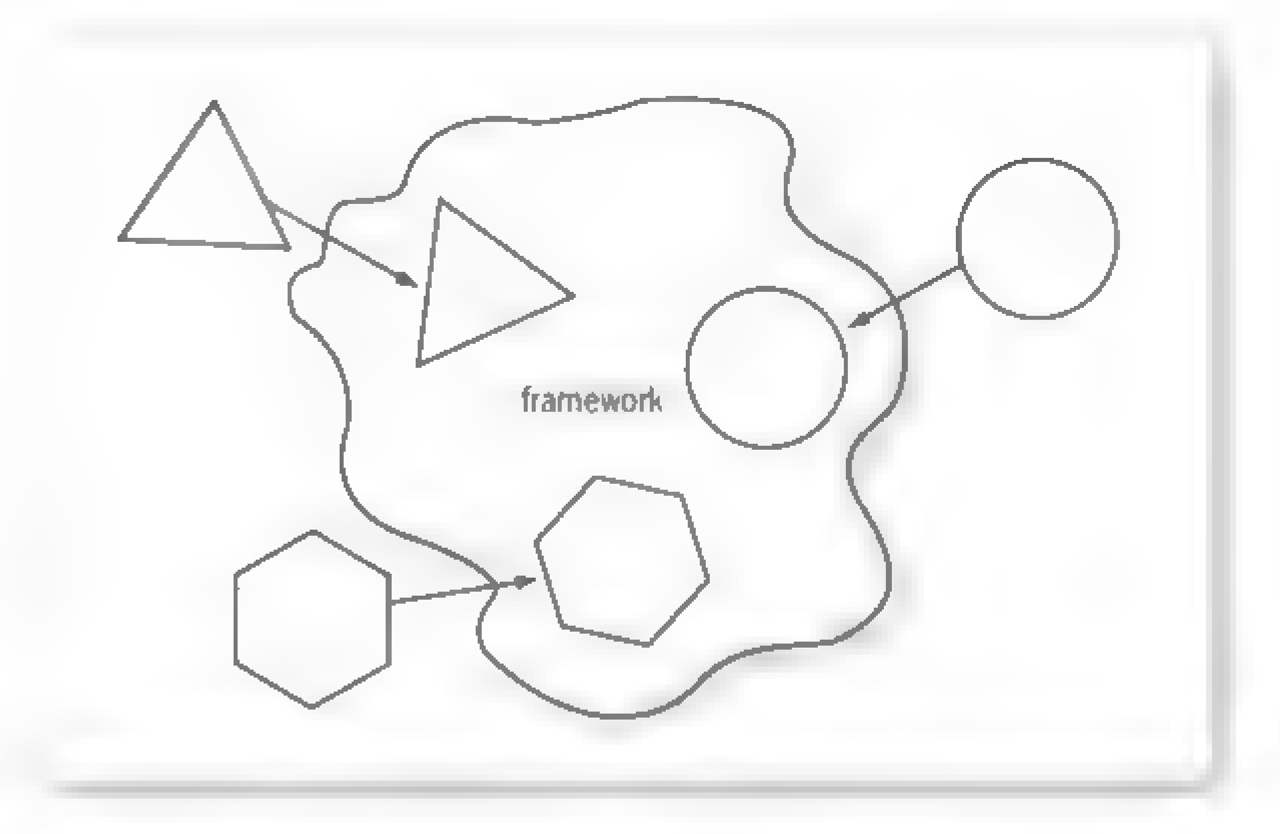


Figura 3. Relación entre un framework y las partes de nuestro código.

La inversión de control es la diferencia clave que existe entre una librería y un framework. Cuando usamos una librería, nos encargamos de manejar el ciclo de vida de sus objetos y de activarlos (utilizarlos). En cambio, en un framework, el que está en control de la ejecución es este, y se encarga de activar nuestros objetos cuando lo cree necesario.

Todos los frameworks tratan sobre la inversión de control, aunque, popularmente, solo se utiliza este concepto para tratar los frameworks de inyección de dependencia o de configuración de objetos. Como dijimos, la inyección de dependencias trata sobre cómo configurar los colaboradores de un objeto. Debemos hacerlo aunque no utilicemos un framework específicamente para esto. Lo que necesitamos son uno o más objetos que se encarguen de crear las dependencias, crear el objeto que nos interesa y enlazarlos.

Los frameworks de inyección de dependencias no solamente manejan la creación y configuración de los objetos y sus dependencias, sino que agregan más funcionalidades que permiten establecer cuándo y cómo se deben crear (o, si es posible, reutilizar instancias para no crear objetos costosos de más).

Hay que considerar que no todos los colaboradores deben ser externalizados. Solamente deberíamos tener como dependencias a aquellos colaboradores que existan como concepto fuera del objeto y que no están atados al ciclo de vida de este. Por ejemplo, el mes de febrero existe como objeto más allá de si es colaborador para la fecha 15 de febrero de 2014.
En cambio, una colección para guardar los elementos que componen una figura debería permanecer oculta dentro del objeto figura, ya que es un detalle de implementación que no le interesa a un cliente. Veamos cómo serían ambos ejemplos en código. Primero, el caso en el que el colaborador es externo y no depende de nuestro objeto para existir:

```
public class Fecha {
  ...
  private final Mes mes;
  ...
  public Fecha(final Dia dia, final Mes mes, final Año año) {
    ...
  }
  ...
}
```

Se apunta a crear protocolos que permitan ser encadenados y que puedan formar frases que se parezcan lo más posible a enunciados del lenguaje humano. De esta forma, es mas natural pasar del dominio al modelo: un ejemplo sería Punto conX(x).yConY(y) para construir un punto. Esto es claro, pero se requieren protocolos más complicados para armar las frases

En el segundo caso, el colaborador es interno y solamente existe mientras está vivo el objeto que lo contiene:

```
public class Figura {
  private List<Forma> formas = new ArrayList<Forma>();
  ...
  public Figura() {
    ...
  }
  ...
  public void agregarForma(final Forma forma) {
    formas().add(forma);
  }
  ...
}
```

Creación de objetos

Vimos que debemos delegar la responsabilidad de creación y configuración de los objetos en otros; ahora, avancemos un poco más en las distintas estrategias para esta tarea. Existen varios patrones aceptados por la comunidad de programadores sobre cómo enfrentar el problema de

VÁLIDO POR NATURALEZA

Supongamos que utilizamos un int para modelar algo que siempre debe ser positivo: debemos usarlo en todos los lados donde un parámetro sea positivo, una y otra vez. En su lugar, podríamos crear un nuevo tipo que naturalmente represente a los números positivos. De esta manera, no ensuciaremos el código con validaciones repetitivas y abstraeremos el nivel de nuestro programa.

la creación. Los patrones de diseño son soluciones ampliamente aceptadas para problemas recurrentes en los sistemas. Para que surja un patrón, este debe estar validado por varios sistemas que hayan solucionado un problema por medio de este. Existe un amplio catálogo de patrones conocidos, que atacan distintos tipos de problemas.

Método factoría

El primero de estos patrones ya lo hemos visto y usado varias veces en los ejercicios, y es el que está basado en utilizar un método que cree el objeto que corresponde. Este método puede ser tanto de instancia como estático, y permite que el cliente no sepa efectivamente qué implementación se está utilizando. Es el método el encargado de decidir cuál de las implementaciones corresponde. Si es de instancia, es posible construir jerarquías que sobrescriban el comportamiento de este según convenga.

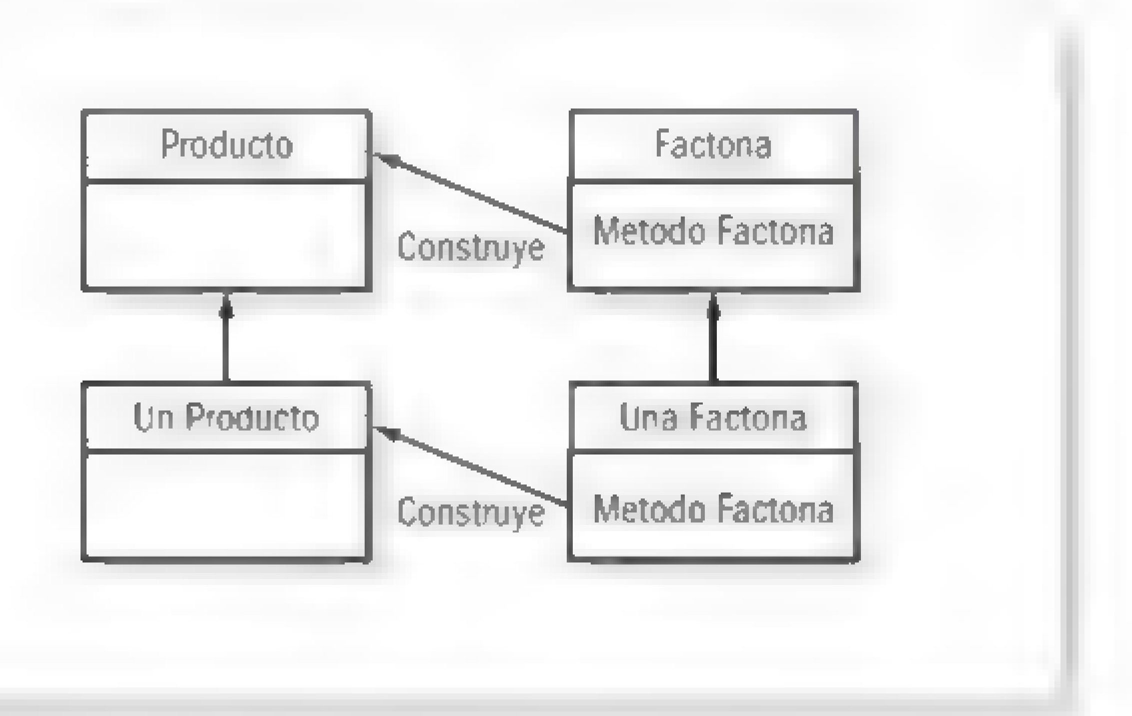


Figura 4. Diagrama teórico de clases.

La clase Collections de Java es un buen ejemplo de este tipo de patrón, donde los distintos métodos crean objetos de los cuales no sabemos a ciencia cierta qué implementaciones tienen:

```
public class Collections {
  ...
  public static <E> List<E> emptyList() {
    ...
  }
  ...
  public static <E> List<E> singletonList(E o) {
    ...
  }
}
```

Factoría abstracta

Supongamos que tenemos una familia de objetos para crear y hay varias implementaciones de cada una. Por ejemplo, imaginemos que tenemos muebles para una sala: sillas, mesas, sillones y lámparas. Además, contamos con distintos estilos para estos muebles: clásico, retro y moderno. Entonces, tenemos la jerarquía de muebles y estilos, y queremos obtener muebles de un mismo estilo de una forma en que no tengamos que especificarlo siempre. Para ello, generaremos otra jerarquía de objetos que nos devuelva las sillas, mesas, sillones y lámparas que corresponden a ella:

ELEGIR EL FRAMEWORK A UTILIZAR

Si queremos utilizar un framework de inyección de dependencias debemos investigar y ver qué requiere cada uno de nuestros objetos para poder crearlos y configurarlos. Si nos fuerza a implementar tal o cual interfaz o a seguir un estilo que no nos gusta, rechacémoslo. Encontraremos infinidad de estos frameworks y, seguramente, habrá uno que se acomode a nuestro estilo y a nuestras necesidades.

```
...
public void decorarCon(Estilo estilo) {
  setMesa(estilo.mesa());
  setSillas(
    estilo.silla(),
    estilo.silla(),
    estilo.silla(),
    estilo.silla()
  );
  setUnSillon(estilo.sillon());
  setOtroSillon(estilo.sillon());
  ...
}
...
```

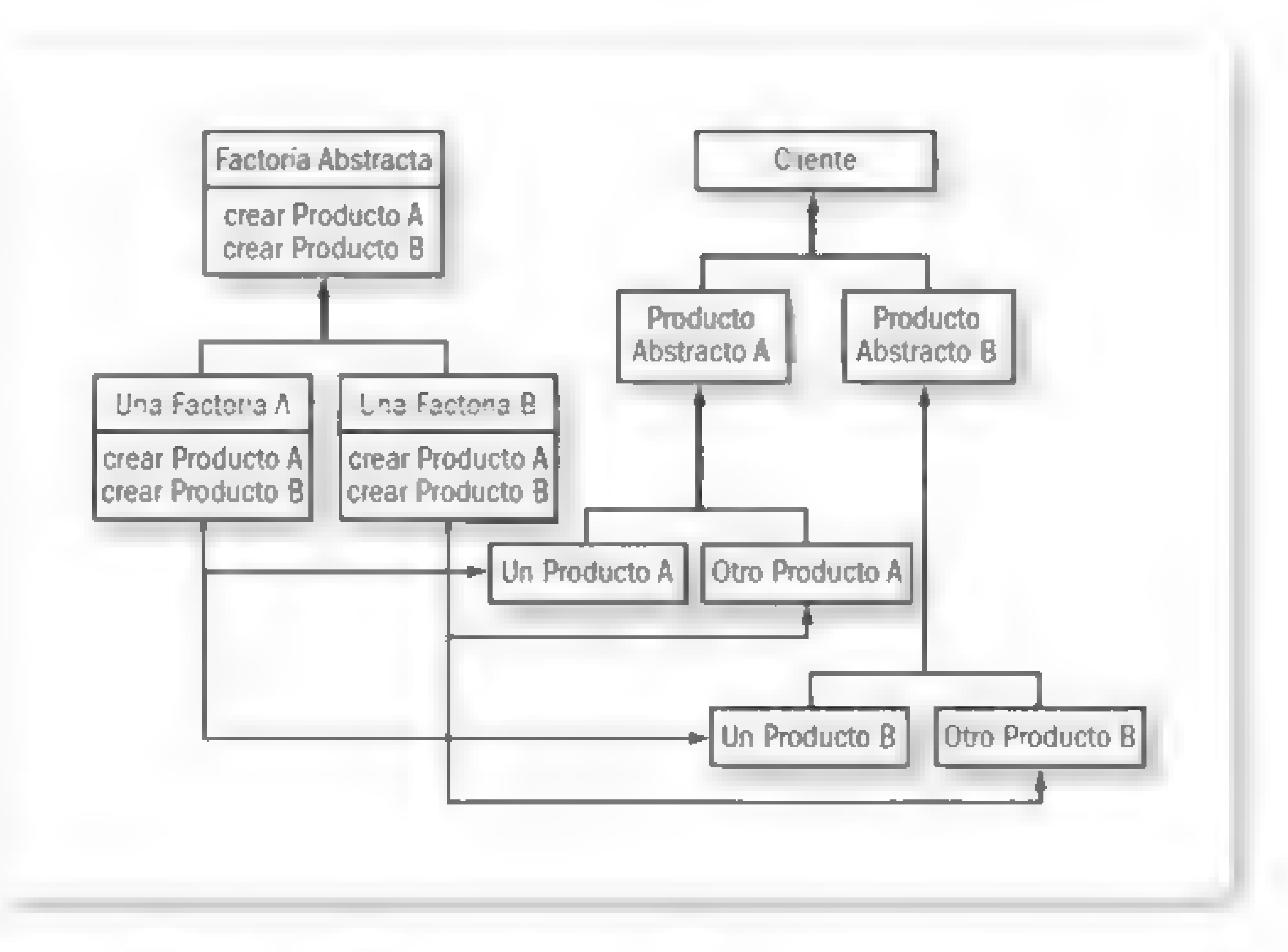


Figura 5. Diagrama teórico de clases de varias factorías con varios productos.

De esta manera, cuando queremos todos los muebles de un mismo estilo solamente tenemos que utilizar el objeto constructor correcto.

Uso de null

La utilización de null presenta un problema No se trata de un objeto, ya que si tratamos de utilizar una referencia que está apuntando a null obtendremos un error. Entonces, lo más conveniente es reducir su uso. ¿Qué significa esto? Por ejemplo, si tenemos una colección vacía, no deberíamos usar null, sino "una colección vacía", incluso en métodos de búsqueda. De esta manera, el cliente no tiene que preguntar si la respuesta es null o no.

NULL SUELE PRESENTAR PROBLEMAS, POR LO CUAL CONVIENE REDUCIR SU USO

También es importante no tener atributos nulos, ni parámetros de más que tengan que ser pasados como null cuando son opcionales. Muchos métodos que reciben varios parámetros tienden a permitir que algunos sean nulos, lo que complica el código.

Es conveniente agregar métodos con la cantidad de parámetros obligatorios correcta. Si diseñamos correctamente, podremos disminuir el uso de null y el consecuente código para preguntar si algo es null o no.

SOBRE LOS PATRONES DE DISE

Los patrones de diseño están basados en los estudios realizados por el arquitecto Christopher Alexander a finales de la década del 70. Alexander propuso una forma de reutilizar conceptos para construir a cualquier escala, para lo cual se basó en la observación de muchas ciudades y edificios, de los que abstrajo las formas básicas y comunes. Luego, la computación tomó estas ideas y las aplicó a su propia disciplina

```
File directorioPadre    directorio.getParentFile();
// Tengo que preguntar siempre
if(directorioPadre != null) {
// Hacemos algo con el directorio padre
}
```

Todo código cliente de este tipo de método tiene que preguntar indefectiblemente si es null o no. Sería mucho más fácil mover ese código a la clase correspondiente y que los clientes solamente provean el comportamiento específico, o utilizar un iterador y usar foreach.

```
// Código hipotético
for(File padre : directorio.getParentFile()) {
// Hacemos algo con el directorio padre
}
```

Utilizar foreach es práctico, ya que no se ejecuta si no hay nada. También podemos pasar una clase anónima a un método que aplique dicha clase al objeto que puede ser nulo, y así verificar si es nulo o no está en un solo lugar.

```
directorio.getParentFile(newWith<File>() {
 @Override public void do(File parent) {
 // Hacemos algo con el directorio padre
 }
});
```

Otra opción es modelar el vacío, el no objeto, explícitamente, lo que se conoce como patrón Null Object. De esta forma, creamos un objeto polimórfico que representa la nada, con el tipo de objetos que necesitamos que sea así.

```
public class CuentaNula extends Cuenta {
  @Override
  BigDecimal saldo() {
    return BigDecimal.ZERO;
  }

  @Override
  void transferirA(Cuenta destino, BigDecimal monto)
  {
    return;
  }

  @Override
  Cuenta combinarCon(Cuenta otraCuenta) {
    return otraCuenta;
  }
  ...
}
```

Estos objetos nulos deben devolver objetos que representen nulos en todos los métodos que requieran respuesta y, en consecuencia, no hacer nada en todos ellos. Así, la semántica de nulo se va extendiendo por el sistema en vez de arrojar error. Los errores se deberían atrapar con tests unitarios bien hechos.

RESUMEN

Conocimos herramientas que nos ayudarán a diseñar mejores sistemas: comunicarán claramente lo que hacen y serán comprensibles, flexibles y sostenibles. Además, como parte de estas técnicas, hemos visto algunos patrones de diseño funcionales.

Actividades

TEST DE AUTOEVALUACIÓN

1 ¿Por qué los **Java Beans** no son una buena manera de desarrollar?

2 ¿De qué forma aseguraría la inmutabilidad de sus objetos?

3 ¿Cuál es la razón de que sea tan importante el concepto de **inmutabilidad**?

4 Defina **inversión de control**.

5 ¿Qué nos ofrece un **framework de inyección de dependencias**?

EJERCICIOS PRÁCTICOS

1 Modele una fecha como **Java Bean** y resuelva los días del mes de febrero. ¿Qué diferencias hay con el modelo de fecha que encontramos en el ejemplo de código?

2 Modele una lista inmutable.

3 Modifique el ejercicio de capítulos anteriores referido al uso de colecciones inmutables.

4 Modele un subsistema de archivos y directorios sin utilizar null, creando todos los conceptos necesarios. Utilice en el fondo la API de Java.

PROFESOR EN LÍNEA

Si tiene alguna consulta técnica relacionada con el contenido, puede contactarse con nuestros expertos: **profesor@redusers.com**.

Java desde cero

En esta obra haremos un recorrido que comenzará con los conceptos fundamentales de la programación orientada a objetos, el diseño y el desarrollo de software. Todos los procedimientos son expuestos de manera práctica con código fuente de ejemplo, diagramas conceptuales y la teoría necesaria para comprender en profundidad cada tema presentado.

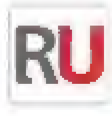
Dentro del libro encontrará*:

Programación orientada a objetos / Sintaxis del lenguaje / Las clases y sus diferentes utilizaciones / Interfaces y enumeraciones / Excepciones y genéricos Librerías base / Anotaciones

* Parte del contenido de este libro fue publicado previamente en Java, de esta misma editorial.

Otros títulos de la colección:

ISBN 978-987-1949-68-7



PROFESOR EN LÍNEA
Ante cualquier consulta técnica relacionada con el libro, puede contactarse con nuestros expertos: profesor@redusers.com.